Monocular 3D Mesh Prediction for Cars in Autonomous Driving Scenarios

Eric Guisado Bandrés

Degree in Mathematics
Degree in Engineering Physics

Supervisor:
Prof. Sanja Fidler

Tutor:
Prof. Xavier Giró

April 2019

Abstract

In the context of autonomous driving cars, a necessity for precise 3D reconstructions of the road and environment is every day more explicit. The present work aims to expose an approach for reconstructing vehicles from a single (RGB) image of a road-like scenario. It is based on previous 3D mesh prediction algorithms that mix Convolutional Neural Network architectures for feature extraction with Graph Convolutional Networks. We build our own dataset and train our model to predict 3D meshes, providing the network with a single RGB image of a car and a 3D bounding box. We reveal how this method recovers precisely most of the geometric details of a car as well as shows successful levels of accuracy.

Keywords: *Machine learning* • *Deep learning* • *Computer vision* • *Autonomous driving* • *3D shape generation* • *Graph convolutional network* • *Mesh reconstruction*

Acknowledgements

I would like to express my very deep appreciation to Professor Sanja Fidler, my research supervisor, for her willingness to host me as a visiting student at her research group, as well as for her enthusiastic direction and useful critiques of this present work. I would also like to thank the University of Toronto for making this stay possible and Vector Institute for providing me with all computational resources and amenities to comfortably carry out the entirety of the project. I would also like to acknowledge Professor Xavier Giró for mentoring my project from Barcelona.

My special thanks are extended to Professor Miguel Ángel Barja, Professor Toni Pascual and Professor Eduardo Alarcón, respectively director, mobility director and deputy director of the Interdisciplinary Higher Education Centre (CFIS), for their excellent work in the management of this Mobility Program, which has facilitated me the opportunity to be hosted in one of the most cutting-edge research institutions of the world.

I wish to acknowledge the help provided by Professor Fidler students Hang Chu and Wenzheng Chen, for their countless observations and useful contributions to this work, as well as to Jordi Fortuny, Rafel Palliser, Dídac Surís and Louis Clergue, fellow students from Polytechnic University of Catalonia that have supported me during the whole project, and which whom I have the pleasure to share a very good friendship. I am particularly grateful to my dear friend and housemate Jordi Fortuny for his endless patience and comprehension throughout the whole stay.

I cannot forget to mention and acknowledge my secondary school teachers Antonio García and David Pi, for putting so much effort to make me discover and realize the intuition and beauty of mathematics.

Lastly, I need to thank my family for their incessant support and affection through my education, even when very long distance has separated us, and especially my sister for always being an inspiration for me with her limitless strength and creativity.

Contents

Introduction

Numerous efforts have been put recently to boost the development of Autonomous Driving. Indeed, over 70% of the approximately 5 billion dollars invested in auto technologies startups in 2018 were committed to autonomous vehicle companies [2].

It is evident that Artificial Intelligence plays a major role in the evolution of self-driving cars. It is fundamentally present in the main tasks necessary to achieve a feasible road situation with autonomous driving cars. These tasks can be summarized as essentially two major jobs: road and environment recognition; and vehicle control. This means, on the one hand, the rendering and reconstruction of road's key elements and geometry from the data provided by car sensors (monocular cameras, stereo cameras, 3D laser sensors or LIDAR, etc) and, on the other, interpreting how to use this information to perform each possible action while ensuring that legality is respected as well as a safe driving is warranted.

There is a whole bunch of tasks underlying prosperous road-scene understanding, which include object detection, depth prediction or instance segmentation. With the objective of training and testing algorithms, several datasets have been designed to evaluate their performance into benchmarks, being KITTI [7] [17] one of the most popular. However, these datasets usually possess a reduced amount of data because of the expensiveness and complexity of its annotation. Therefore, most of the approaches are forced to opt for synthetic datasets to be able to train with a reasonable number of examples. At the same time, more and more resources are being nowadays dedicated to promote and assist the annotation of data collected from car sensors.

This report aims to exhibit our developed approach for generating 3D shapes for cars from a single RGB image. As a possible application, this could be a powerful tool to help and boost the annotation of 3D geometry of an autonomous driving scenario. We address this problem with a method derived from a robust end-to-end framework called Pixel2Mesh. With this system, we are able to reconstruct 3D meshes with accurate 3D coordinates as well as key geometric details of the car shown in an image, with the aid of a 3D bounding box.

We will first start with a summarized overview of some Computer Vision fundamentals indispensable to understand the contents of the present work. After that, some of the major contributions to the field that have helped us build this method are commented. Then, we will detail all the key pieces that constitute our network, like a very wide synthetic dataset we have rendered to train it. Finally, we will show the main results and metrics as well as discuss weak points and possible applications.

Chapter 1

Computer vision overview

1.1 Introduction

Images are 2D projections of the 3D world, and that unavoidably implies that images contain less information than the 3D scenes behind them, since one of the dimensions is unconstrained. This does not stop, however, human brain from being able to interpret and reconstruct not only the geometrical structure of the scenario captured, but also identify objects, fill missing spots of the image or even foretell people's emotions from their facial gestures. Even though visual system can also be fooled to build a wrong representation of the surrounding environment (see figure 1.1), it is still unbelievably good in most scenarios, and that is the reason why perceptual psychologists have spent decades trying to understand how the visual system works [25].

Almost at the same time as digital cameras were developed, images became handy mathematical objects. This facilitated that researchers in computer vision started developing techniques to mimic or even improve human brain capacity for understanding and reconstructing 3D scenes from digital images. Nowadays, computer vision techniques are already outperforming humans in certain tasks, in which most of the time a network receives only an RGB image and it is capable of classifying objects or species, recognizing people from facial gestures, etc.

Figure 1.1: Optical illusion floor at Britain's Casa Ceramica, which tricks our visual system making us perceive a curved floor.

This chapter aims to give a brief overview of some generalities about computer vision and its fundamental tools.

1.2 Image formation

In this section we are going to focus on introducing the basic tools necessary for describing the geometry of a 3D scene and describing the mathematical relations that lead to the image formation process.

1.2.1 Projection

There are different models to describe projections of 3D space onto a plane. We must mention the four most commonly used: the *orthographic* projection, the *scaled-orthographic* projection, the *para-perspective* and the *perspective* projection. Although the first model is the simplest one, perspective is actually the most widely used projection in computer graphics and computer vision, since it is the one that more accurately emulates the behaviour of real cameras.

Let us start describing an orthographic projection, also known as *orthography*. Consider a three-dimensional point $\boldsymbol{p} = [x, y, z] \in \mathbb{R}^3$. This point can be written in homogeneous coordinates as $\tilde{\boldsymbol{p}} = [x, y, z, 1]$. Using these coordinates, orthography simply consists in dropping the z component of $\tilde{\boldsymbol{p}}$. This process leaves three coordinates representing the homogeneous coordinates of a 2D point. Let $\tilde{\boldsymbol{x}}$ be the homogeneous coordinates of the 2D projected point. We can write:

$$\tilde{\boldsymbol{x}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tilde{\boldsymbol{p}}. \tag{1.1}$$

What makes the orthographic projection the simplest model is the fact that we can write it using inhomogeneous coordinates, without any normalization needed. Namely, in equation (1.1) we can drop the last coordinate and obtain the inhomogeneous coordinates equation: $\boldsymbol{x} = [I_{2\times 2}|0_{2\times 1}]\boldsymbol{p}$. Note that an orthography is equivalent to projecting each three-dimensional point orthogonally onto the XY plane.

However, in practice, world coordinates might be meters whereas image sensors physically measure millimeters (and later pixels), so it is necessary to scale world coordinates to fit onto the image sensor [25]. This is the reason why usually *scaled-orthography* is used:

$$\tilde{\boldsymbol{x}} = \begin{bmatrix} s & 0 & 0 & 0 \\ 0 & s & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tilde{\boldsymbol{p}}. \tag{1.2}$$

This projection successfully models long focal length lenses and objects that are shallow compared to their distance to the camera.

A slightly similar model can be obtained by, instead of projecting points orthogonally onto the XY plane, projecting them parallel to the line of sight to the object center onto a hypothetical plane parallel to the XY plane. The resulting projected set of points is later scaled as happens in orthographic projections. This process is the so-called *para-perspective* projection or, simply, *para-perspective*.

The most accurate and widely used projection modelling image formation is 3D *perspective*. A perspective can be written in homogeneous coordinates simply as:

$$\tilde{\boldsymbol{x}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tilde{\boldsymbol{p}}. \tag{1.3}$$

Observe that this is equivalent to dropping the w coordinate. Now, if $\boldsymbol{p} = [x, y, z]$, in inhomogeneous coordinates this reads as $\boldsymbol{x} = [x/z, y/z]$.

1.2.2 Camera intrinsics

So far we have a model describing how 3D quantities are projected into 2D. However, the resulting coordinates must be still transformed to match the image sensor spacing and relative position. Image sensors return pixel values indexed by integer pixel coordinates $[\bar{x}, \bar{y}] \in \mathbb{Z}^2$. The most common convention is that pixel coordinates increase its value moving down and to the right.

The full relation between the 3D point coordinates and the sensor coordinates is the following: pixel coordinates are a discretization of the sensor plane coordinates $x_s = s_x\bar{x}$, $y_s = s_y\bar{y}$, where s_x and s_y express the conversion of integers to physical sensor units along x and y axes.

Moreover, these represent coordinates relative to the sensor, which is not located or oriented at the same place that the camera center. Therefore, there exists a rigid transformation $T_s \in SE(3, \mathbb{R})$ of the sensor system of coordinates into the camera system of coordinates. This transformation can be always decomposed into a translation of the origin plus a proper rotation of the axes, that is, we can write $T_s(\boldsymbol{x}) = R_s\,\boldsymbol{x} + \boldsymbol{t}_s$ with $\boldsymbol{t}_s \in \mathbb{R}^3$ and R_s an orthogonal matrix, with $\det R_s = 1$. In terms of homogeneous coordinates T_s can be represented by the following 4×4 entries matrix:

$$T_s = \begin{bmatrix} & R_s & & \boldsymbol{t}_s \\ & & & \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{1.4}$$

Putting it all together we have the following equation:

$$\tilde{\boldsymbol{p}} = \begin{bmatrix} & R_s & & \boldsymbol{t}_s \\ & & & \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \bar{x} \\ \bar{y} \\ 1 \end{bmatrix} = M\tilde{\boldsymbol{x}}. \tag{1.5}$$

Matrix $M \in \mathbb{R}^{4\times4}$ is the so-called *camera model* matrix. By construction, we see that M is defined by 8 parameters: 3 parameters describing a rotation matrix (angle of rotation along each axis), 3 parameters defining a translation and the two scaling factors s_x and s_y. Hence,

we have a total of eight unknowns. Nevertheless, estimating a matrix with seven degrees of freedom is rarely done in practice, where, in contrast, a general 3×3 matrix is assumed [25].

Note that if we drop the last row of M we obtain directly the inhomogeneous coordinates of the 3D point up to a scale. This also defines a inverse relation between sensor pixel homogeneous coordinates $\tilde{\boldsymbol{x}}$ and inhomogeneous 3D coordinates $\boldsymbol{p}$:

$$\tilde{\boldsymbol{x}} = M^{-1}\boldsymbol{p} = K\boldsymbol{p}. \tag{1.6}$$

Here, K is the so-called *calibration matrix*, also known as *camera intrinsic parameters* or, simply, *camera intrinsics*. Even though, as already discussed before, this matrix has seven degrees of freedom, in practice it is considered to have eight. Nevertheless, most popular computer vision books treat K as an upper-triangular matrix with five degrees of freedom, e.g., [9]. It is usually written in the following fashion:

$$K = \begin{bmatrix} f_x & s & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}. \tag{1.7}$$

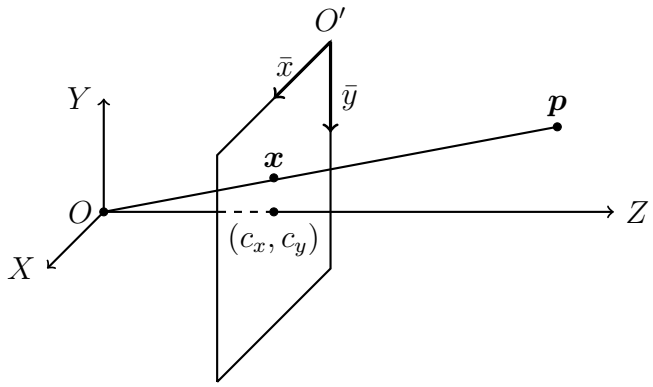


Figure 1.2: Simplified camera model. Please note that two steps are involved: projection of the 3D point $\boldsymbol{p}$ onto a 2D plane and scaled transformation of camera reference frame (O) into sensor pixel coordinates reference frame (O').

Here f_y and f_x are the so-called *focal lengths*, which cannot be zero because, otherwise, K would have rank lower than 3 and we would be projecting onto a line or a point instead of on a plane. It makes, thus, perfect sense to talk about an *aspect ratio* $a := f_y/f_x$. Parameters c_x and c_y represent the *optical center* pixel coordinates, and s models the effects of a possible *skew* between the sensor axes. In practice, a simpler form can be used for many applications, which assumes aspect ratio 1 and $s = 0$:

$$K = \begin{bmatrix} f & 0 & c_x \\ 0 & f & c_y \\ 0 & 0 & 1 \end{bmatrix}. \tag{1.8}$$

1.2.3 Camera extrinsics

In practice, however, 3D coordinates are not measured in the camera system of coordinates, but in a different reference frame. Both systems of coordinates are, again, related by a euclidian transformation $T \in SE(3, \mathbb{R})$, which can be decomposed into a rotation R plus a translation $\boldsymbol{t}$. These form the *camera extrinsic parameters* or *camera extrinsics*. Finally, we obtain an equation relating pixel homogeneous coordinates and reference frame 3D coordinates $\boldsymbol{p}_r$:

$$\tilde{\boldsymbol{x}} = KT\,\boldsymbol{p}_r = P\,\boldsymbol{p}_r \tag{1.9}$$

P is known as the *camera matrix*, and it is the matrix we can actually calibrate based only on external measurements. If we wanted to estimate the camera intrinsics we would rather have to first estimate T (based on a calibration between the camera system of coordinates and the reference system of coordinates) and later compute the unknown entries of K.

1.3 Neural networks

A *neural network* is a computer system that attempts to mimic how biological neurons work. Although the comparison is limited, it is really the underlying idea behind the concept of neural networks. Just as biological neurons constitute the basic computational unit of the brain, neural networks are formed by (artificial) *neurons* or *linear units*.

1.3.1 Neurons or linear units

Brain neurons are connected through *synapses*. They receive signals from their *dentrites* and output signals along their *axons*. This is modelled computationally as a function h that receives several inputs $\boldsymbol{x} = \{x_1, \dots, x_n\}$ (from its "dendrites") and outputs a positive signal $h(\boldsymbol{x}) \in \mathbb{R}_{\geq 0}$. This function is made up of a linear function followed by an *activation function*. The linear function simply weights which inputs are more important, in order to strength particular synaptic connections. If the weighted sum of this signals is above a certain threshold, the neuron can *fire*, and this is done by means of an activation function.

To be precise, let $\boldsymbol{w} = \{w_1, \dots, w_n\} \subset \mathbb{R}$ the linear coefficients and $b \in \mathbb{R}$ the bias of the linear function of our neuron. This linear function can be written as

$$y = \boldsymbol{w} \cdot \boldsymbol{x} + b = \sum_i w_i x_i + b. \tag{1.10}$$

This is followed by an activation function $f(y) = f(\boldsymbol{w} \cdot \boldsymbol{x} + b)$ that fires if y is greater than a certain threshold. This threshold, however, can be set to zero by choosing a suitable bias b. There are several activation functions that are commonly used:

(a) *Sigmoid*. Sigmoid non-linearity can be expressed as $\sigma(y) = 1/(1 + e^{-y})$. Although it is the one that more accurately mimics a firing rate of a neuron, it is infrequently used as an activation function.

(b) *Hyperbolic tangent*. It can be written as $f(y) = \tanh y$. In practice, it is always preferred to the sigmoid activation function.

(c) *Rectified linear unit* (ReLU). It is also known as the *ramp function*, and consists in thresholding the input at zero, that is, computing $f(y) = \max(0, y)$. This is a very easy computation that was found to accelerate convergence. Nevertheless, there is a downside, which that neurons can easily die and become unusable if $\boldsymbol{w} \cdot \boldsymbol{x} + b < 0$ for all $\boldsymbol{x}$ in one optimization step. If that is the case, gradient will vanish and those parameters ($\boldsymbol{w}$ and b) will never be updated.

(d) *Maxout.* Maxout was introduced by Ian Goodfellow in [8], where he proposed to compute the output of each neuron as $\max(\boldsymbol{w}_1 \cdot \boldsymbol{x} + b_1, \boldsymbol{w}_2 \cdot \boldsymbol{x} + b_2)$. ReLU and Leaky-ReLU can be thought as particular cases of a maxout. The drawback of this activation function is that it doubles the number of parameters of each unit.

1.3.2 Neural network architectures

As commented above, neural networks are build from linear units. Indeed, they can be formally presented as a directed graph whose nodes are linear units and whose edges connect outputs of neurons as inputs to other units. To avoid cycles (which would imply infinite loops), neural networks are usually organized into layers.

More often, each unit of a layer is connected to all units of the previous layer. In that case we say we have a *fully-connected layer*. We can have as many layers as we wish, each with a different number of units. By convention, the first layer layer is called the *input layer*, the last *output layer* and the remainder are *hidden layers*. For example, let us consider we are willing to model the relationship between two quantities y_1 and y_2, as a function of three variables x_1, x_2 and x_3. Then $\boldsymbol{x} = \{x_i\}$ would be the input layer, $\boldsymbol{y} = \{y_j\}$ the output layer, and then we could add one hidden layer of four units (see figure 1.3). This model might result to be too simple to explain it and it that case we would add more hidden layers.

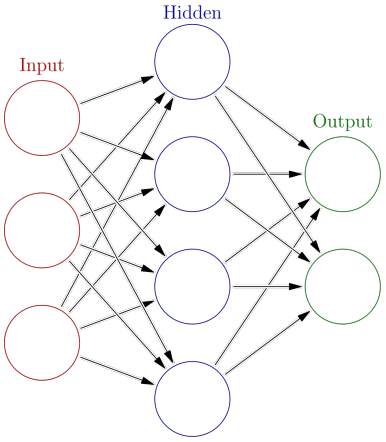


Figure 1.3: Example of neural network architecture with one hidden layer.

1.3.3 Neural networks optimization

Neural networks are used to model complex functions describing relations between variables: $\boldsymbol{y} = f(\boldsymbol{x})$. This function is, by construction, the result of composing multiple linear units, each of which has some weight parameters and a bias. The general goal is to adjust the whole set of parameters, let us call it Θ, so that $f = f_\Theta : \mathbb{R}^n \to \mathbb{R}^m$ performs "well" in a certain task. In order to tune these parameters we first fit the model to perform well on a certain set of samples that we call the *training set*, in a process that is called *training*.

To measure how well a neural network does in a certain task it is necessary to define a *loss function*, that is, a function that measures the unhappiness of a given output of the network. More precisely, let $\boldsymbol{y} = f_\Theta(\boldsymbol{x}) \in \mathbb{R}^m$ be the output of the neural network for a given input $\boldsymbol{x} \in \mathbb{R}^n$, and let $\hat{\boldsymbol{y}}$ be the output corresponding to $\boldsymbol{x}$ in our training set (usually referred as a *ground truth* output). Then the loss function it is in general a function

$$\begin{aligned} L_\Theta : \mathbb{R}^m \times \mathbb{R}^m &\longrightarrow \mathbb{R} \\ (\boldsymbol{y}, \hat{\boldsymbol{y}}) &\longrightarrow L(\boldsymbol{y}, \hat{\boldsymbol{y}}) \end{aligned} \tag{1.11}$$

that maps each pair of *predicted* output $\boldsymbol{y}$ and ground truth output $\hat{\boldsymbol{y}}$ to a real number, representing the "cost" or unhappiness of the prediction based on the actual output.

Training a network consists in minimizing this objective function for the samples found in the training set. This is usually achieved using numerical optimization algorithms. Most of them are based on gradient descent: every parameter of the network is updated taking steps in the opposite direction of the gradient with respect to that parameter, that is,

$$\theta := \theta - \lambda \frac{\partial L_\Theta}{\partial \theta}, \tag{1.12}$$

for all $\theta \in \Theta$. λ is the so-called *learning rate*. The derivative with respect to each parameter is computed recursively by means of the chain rule: consider two linear units from adjacent layers. Let $\boldsymbol{w}, b$ be the weights and bias of the first and assume we know the gradient of the loss $\frac{\partial L}{\partial y}$ with respect to the input y of the second unit (which is also the output of the first unit). Then, by virtue of chain's rule,

$$\frac{\partial L}{\partial w_i} = \frac{\partial y}{\partial w_i}\frac{\partial L}{\partial y}, \quad \frac{\partial L}{\partial b} = \frac{\partial y}{\partial b}\frac{\partial L}{\partial y}, \quad \frac{\partial L}{\partial x_i} = \frac{\partial y}{\partial x_i}\frac{\partial L}{\partial y}. \tag{1.13}$$

Now, $y = f(\boldsymbol{w} \cdot \boldsymbol{x} + b)$, where f is the activation function, so partial derivatives of y with respect to $\boldsymbol{w}$, b and $\boldsymbol{x}$ are known and easy to compute. Once $\partial_{\boldsymbol{x}} L$ is known, one can repeat the same process again for each unit of the layer before the first unit. This recursive algorithm, named *back-propagation* [19], dramatically simplified gradient computation and, ultimately, gradient descent optimization for neural networks.

Training a network with a loss function that does not explicitly depend on the ground truth output, that is, $\frac{\partial L}{\partial \hat{y}_j} \equiv 0$, $j = 1 \div m$, is called *unsupervised learning*. Otherwise it is referred to as *supervised learning*.

1.3.4 Regularization

Sometimes the adjustment of model parameters to the training set of samples is too strong that its performance over new samples (*inference*) is poor. In this case we say the network has experienced *overfitting*. Overfitting generally means that the network has learnt too much from the training set that it is only able to produce good results for inputs that belong to that set.

There are several techniques used to prevent overfitting, which we call *regularization* techniques. All of them share an interest in reducing the complexity of the function described by the neural network. Here we enumerate some of them:

(a) *L_2 regularization*. This approach faces the problem by encouraging the network to not have large weights that would lead to complex behaviours of the model. This is done by adding a penalty term

$$\tilde{L}_\Theta(\boldsymbol{y}, \hat{\boldsymbol{y}}) = L_\Theta(\boldsymbol{y}, \hat{\boldsymbol{y}}) + \gamma \|\Theta\|_2^2 \tag{1.14}$$

to the loss function, with a fixed weight $\gamma > 0$. This parameter is usually known as *weight decay*.

(b) *Drop out* [23], a technique that consists in randomly dropping units from the neural network during training.

(c) *Data augmentation*. Data augmentation usually stands for effectively increasing the size of the training dataset by adding noise to its samples.

Aside from a training set, it is a common practice to establish two other sets: a *test* set and a *validation* set. On the one hand, the test set serves as a set in which we can test our model and see how well it does perform. On the other hand, the validation set serves to assure that improvements made on fitting the train data are reflected in a better performance on new data, so that we do not develop overfitting.

1.4 Convolutional neural networks

Neural networks form really powerful tools that allow great results in inference on highly diverse problems and kinds of data. Nevertheless, images structure encodes many properties that relate to the 3D properties of objects captured on them. As we have seen previously in this chapter, images are structured in pixels, which ultimately represent a discretization of a field of view along the orthogonal axes to the camera. Even though this discretization does not preserve all 3D constraints since it reduces the space from three to two dimensions, it does preserve relations that happen along the camera plane, that is, along X and Y axes.

This observation leads us to the conclusion that by treating images as any other kind of data arrays we are losing the geometric constraints captured on the camera plane. This is one of the reasons why regular neural networks lack of capacity of analyzing successfully scenes represented on images. Moreover, as the resolution of images increases the number of weights and biases that constitute a neural network adds up really quick, and leads to too complex models, that easily tend to overfit or are very hard to train.

Convolutional neural networks are capable, however, of taking advantage of images structure. A convolutional neural network (CNN) still possesses units with learnable weights and biases, which perform dot products that are optionally followed by non-linearities. However, it assumes that the input is an image, so each layer of a convolutional neural network supposes that the input is an image-like data array: $\boldsymbol{x} \in \mathbb{R}^{w \times h \times c}$, where h and w are height and width (the *spatial dimensions*), and c is the *number of channels*. For an RGB image, we would have three channels ($c = 3$), each encoding a different color, but for a grayscale image we would have only one ($c = 1$).

Therefore, the input to a convolutional neural network is a three-dimensional tensor. Each layer of a CNN transforms a three-dimensional tensor into a three-dimensional tensor, which may have different number of channels or different spatial dimensions than the input. These layers might or not have learnable parameters.

1.4.1 Convolutional neural network layers

There are several types of layers used in convolutional neural networks. These are by far the most popular:

1. *Convolutional layer* (`conv`). A convolutional layer computes an output tensor whose entries are obtained each as a dot product between their weights and a small region around each input tensor entry.

2. *Fully-connected layer* (`fc`). As a regular neural network fully-connected layer, each unit is connected to all entries of the previous tensor.

3. *ReLU layer* (`relu`). A *rectified linear unit* applies an activation function, like $f(x) = \max(0, x)$, to each entry of the input.

4. *Pooling layer* (`pool`). A pooling layer downsamples the input tensor by reducing the spatial dimensions (w and h).

A typical CNN can be shortly summarized as a list of these layers. For example: `conv - relu - conv - relu - pool - conv - relu - pool - fc`.

Convolutional layer. A convolutional layer consists of a stack of learnable filters. Each filter or *kernel* consists of a set of weights arranged as a three-dimensional tensor with the same number of channels than the input, but typically with small spatial dimensions (e.g., $h = w = 3$). Filters are usually square, $k = h = w$, with k, the *receptive field*, an odd integer. This filter is slid along each of the spatial dimensions of the input tensor, performing a dot product of the occupied region of the input tensor and the weights of the filter. As a result of this operation we obtain a two-dimensional tensor (a *map*) formed by the dot products at each possible position we slide the filter across. If we stack several filters with same spatial dimensions together, we obtain at the output a stack of maps, that is, a three-dimensional tensor.

More formally, let $\boldsymbol{x} \in \mathbb{R}^{w \times h \times c}$ be the input to our convolutional layer and let us consider a kernel $\boldsymbol{k} \in \mathbb{R}^{k \times k \times c}$ with height and width k, $k \equiv 1 \bmod 2$. Then, the output of this convolution is a new tensor $\boldsymbol{y} \in \mathbb{R}^{w' \times h'}$:

$$\boldsymbol{y}(i,j) = \sum_{l=1}^{c} \sum_{-\lfloor \frac{k}{2} \rfloor \leq i', j' \leq \lfloor \frac{k}{2} \rfloor} \boldsymbol{k}(i', j', l) \cdot \boldsymbol{x}(i + i', j + j', l). \tag{1.15}$$

Note that not all values of i and j might be valid indexes. Indeed, it is easy to see that:

$$1 + \left\lfloor \frac{k}{2} \right\rfloor \leq i \leq w - \left\lfloor \frac{k}{2} \right\rfloor, \quad 1 + \left\lfloor \frac{k}{2} \right\rfloor \leq j \leq h - \left\lfloor \frac{k}{2} \right\rfloor. \tag{1.16}$$

Thus, after each convolutional layer the spatial dimensions will be reduced. This might be an inconvenient on occasion. A widely used solution to face up this problem consists in

increasing the size of spatial dimensions by adding zeros around the border (*zero-padding*). With a suitable selection of the number of zero-padding rows and columns we can fix the output dimensions to the desired ones. Examination of equation (1.16) concludes that to preserve initial spatial dimensions in a convolutional layer we must zero-pad the input with an amount of $\lfloor \frac{k}{2} \rfloor$ rows/columns of zeros each side.

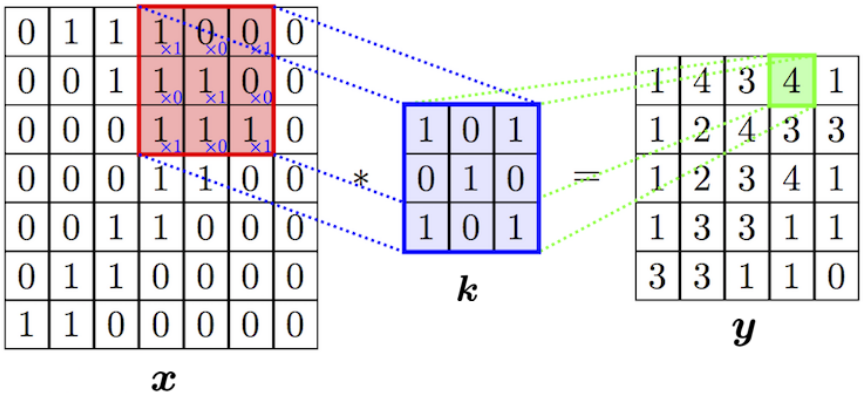


Figure 1.4: Example of convolution of a filter $\boldsymbol{k}$ with an input tensor $\boldsymbol{x}$ of one channel, with stride $s = 1$. Each entry of the output tensor $\boldsymbol{y}$ is obtained as the dot product of kernel weights with the corresponding input tensor region.

Sometimes we may not want to slide the filter across all possible spatial positions of the input tensor, but rather take larger steps. So, we may intend to compute the dot product with entries separated by two positions or three, for instance, instead of one. This separation is called *stride*. A small tweak of equation (1.15) is enough to add this feature to our convolution layer:

$$\boldsymbol{y}(i,j) = \sum_{l=1}^{c} \sum_{-\lfloor \frac{k}{2} \rfloor \leq i', j' \leq \lfloor \frac{k}{2} \rfloor} \boldsymbol{k}(i', j', l) \cdot \boldsymbol{x}(si + i', sj + j', l). \tag{1.17}$$

Note that, again, not all values of i and j will be valid indexes. Indeed, imposing $1 \leq si + i' \leq w$ and $1 \leq sj + j' \leq h$ leads to

$$\left\lceil \frac{1 + \lfloor \frac{k}{2} \rfloor}{s} \right\rceil \leq i \leq \left\lfloor \frac{w - \lfloor \frac{k}{2} \rfloor}{s} \right\rfloor, \quad \left\lceil \frac{1 + \lfloor \frac{k}{2} \rfloor}{s} \right\rceil \leq j \leq \left\lfloor \frac{h - \lfloor \frac{k}{2} \rfloor}{s} \right\rfloor. \tag{1.18}$$

Observe that setting a higher stride drastically reduces the output map spatial dimensions.

Fully-connected layer. As in regular neural networks, fully-connected layers (`fc`) connect to all entries of the input tensor. Therefore, if the input image is a tensor $\boldsymbol{x} \in \mathbb{R}^{w \times h \times c}$, the output of a unit of a `fc` layer is:

$$y = \sum_{1 \leq k \leq c} \sum_{1 \leq j \leq h} \sum_{1 \leq i \leq w} \boldsymbol{w}(i,j,l)\boldsymbol{x}(i,j,l) + b \tag{1.19}$$

Rectified linear unit. A *rectified linear unit* or *rectifier* layer is a layer in which we apply an activation function $f(x) = \max(0, x)$ element-wise to the input tensor, that is, the output tensor is a tensor of the same dimensions in which positive entries are not altered but negative entries are vanished to zero:

$$\boldsymbol{y}(i, j, l) = \max(0, \boldsymbol{x}(i, j, l)). \tag{1.20}$$

Pooling layer. A *pooling layer* is a layer that reduces the spatial dimensions of the input tensor. The most extended form, known as *max-pooling*, halves width and heigh sliding a 2×2 filter across spatial dimensions with stride 2 and preserving only the maximum value, hence its name. Specifically, the output tensor entries are:

$$\boldsymbol{y}(i, j, c) = \max_{0 \leq i', j' \leq 1} x(2i + i', 2j + j'). \tag{1.21}$$

Thus, this layer preserves the number of channels and halves the spatial dimensions.

The goal of introducing these layers periodically in a convolutional neural network is to reduce the amount of parameters needed to perform convolutions with larger receptive field. For example, rather than performing a convolution with kernel size 5×5 we can first pool the input to halve its dimensions and then apply a 3×3 convolution, since the combination of max-pool and 3×3 convolution receptive fields add up to a 5×5 receptive field.

1.4.2 Convolutional neural network architectures

Most convolutional networks are made up from the four kind of layers explained in the previous section: `conv`, `fc`, `relu` and `pool`. These already enable a very wide range of possible architectures. Some of them are very popular and have their own name:

- *LeNet* [16]. LeNet-5 was devolped by Yann Lecun in the late 90's and was designed to automatically classify hand-written digits on American bank cheques. It has a very simple structure, with three convolutional layers, two pooling layers and one fully-connected layer at the end.

- *AlexNet* [15]. This network, designed by Alex Krizhevsky, Ilya Sutskever and Geoff Hinton, was famous after winning the 2012 ImageNet ILSVRC-2012 competition by a margin of 10% over the second place. It consisted of five convolutional layers and three fully-connected layers at the end, with max-pooling after first, second and fifth convolutions. It also introduced *drop out* [23], a technique that consists in randomly dropping units from the neural network during training, after the first two fully-connected layers.

- *ZF Net* [28]. By expanding the size of AlexNet middle convolutional layers and changing the stride and kernel size on the first layer, Matthew Zeiler and Rob Fergus won the ILSVRC-2013 competition. The network is now known as ZFNet, as a short name for Zeiler-Fergus Net.

- *GoogLeNet* [24]. Also known as *Inception-1*, GoogLeNet was developed by Szegedy et al. from Google, and was the winner of ILSVRC-2014 competition. It is a deep convolutional network (22 layers), but is drastically lighter than AlexNet (4 million parameters, compared to AlexNet with 60 million).

- *VGG* [22]. This network developed by Simonyan and Zisserman was the runner-up at the ILSVRC-2014 competition. It is a deep convolutional network of 16 layers and even though it uses a lot of parameters (140 million), it is one of the most popular options nowadays for extracting features from images. Most of the parameters, however, are in the first fully-connected layer, which might be dispensable depending on the task being confronted.

- *ResNet* [11]. *Residual Network*, known by the short name ResNet, was developed by Kaiming He et al. It introduced a very deep architecture (with 152 layers) that used *skip connections* as well as heavy *batch normalization*. On the one hand, skip connections, which consist in adding up the input of a convolutional layer to the output (see figure 1.5, were developed to face up the problem of vanishing gradients that appears after stacking too many layers in a CNN. On the other hand, *batch normalization* [12] is a technique used to increase the stability of a network by normalizing the output of a previous layer (specifically, by subtracting the mean and dividing by the standard deviation).

1.4.3 VGG

The VGG [22] network is one of the most important and popular networks used nowadays. Even though is the runner-up of the ILSVRC-2014 competition and GoogLeNet performed better in the classification task, it is currently the backbone of many models developed at present and it is widely used for tasks like image feature extraction. It was invented by Simonyan and Zisserman, from the Visual Geometry Group from University of Oxford (hence its name).

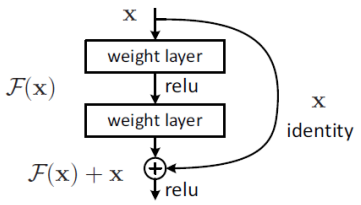


Figure 1.5: Skip connections introduced in [11]. The input is added to the output of a convolutional layer. That way, layer weights are learnt to predict *residuals*.

The reason this network is so popular is that it outstandingly reduced the number of parameters to learn by the network by only using 3×3 filters, discarding larger filters like 7×7 or 11×11, while still maintaining large receptive fields. It also has a very uniform intuitive pattern of layers (see figure 1.6).

To be more exact, when we apply two 3×3 filters, we cover an effective area of 5×5. If we apply three, an effective area of 7×7, and with five an effective area of 11×11. Observe how this leads to a significant reduction of the number of parameters. Indeed there are, for example, $11^2 = 121$ weights in a 11×11 filter whereas five 3×3 kernels only need $5 \cdot 3^2 = 45$ weights. Similarly, a 7×7 kernel needs to learn $7^2 = 49$ parameters but three 3×3 filters only have $3 \cdot 3^2 = 27$ weights. Therefore, this design cuts down the number of parameters

and, thus, the complexity of the network. This has positive effect on preventing overfitting and accelerating optimization convergence.

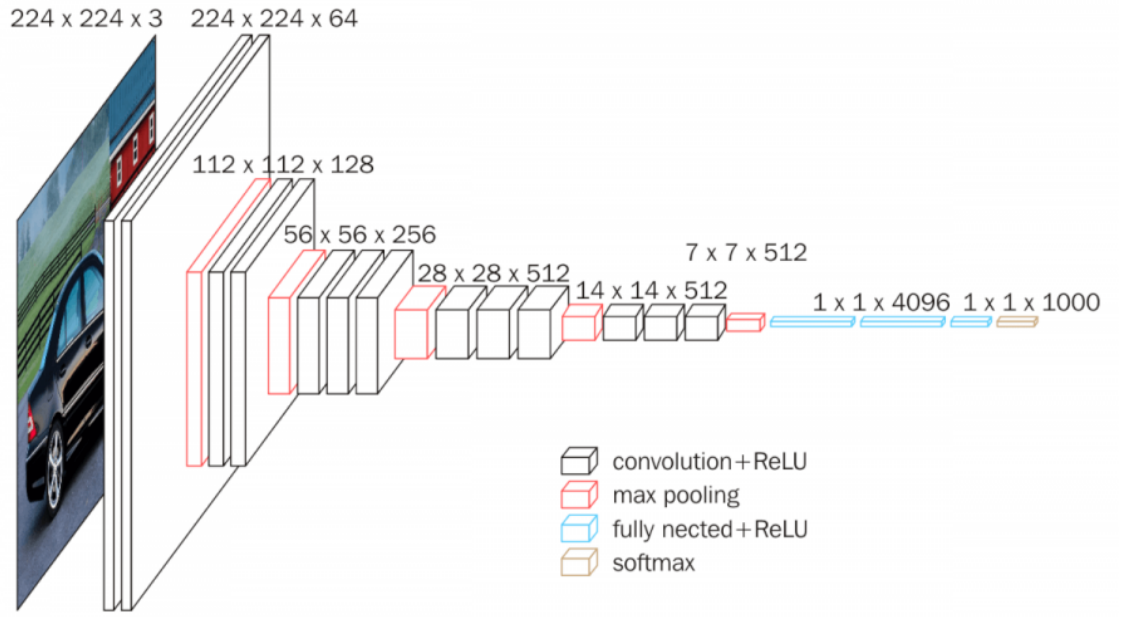


Figure 1.6: VGG-16 architecture used to classify images of ImageNet. Image size is progressively reduced via downsampling, allowing the network to gradually increase the receptive field of its computations.

By the time this network was developed, it performed worse on image classification than GoogLeNet did. However, for the task of image feature extraction this should always be one of the first alternatives. Even though the whole network holds a lot of parameters (140 million), most of them are at the fully-connected layers at the top of the network, which are only used for specific tasks like image classification, but are not necessary for image feature extraction. Moreover, GoogLeNet showed how this fully-connected layers could be substituted using average pooling, drastically reducing the number of parameters in the process.

Chapter 2

Previous work

2.1 Introduction

3D reconstruction of shapes from a single image or multiple views is a pervasive problem in almost every Computer Vision field that aims to recover or modify geometric information of an image. Even if the final goal is not to predict the shape of a certain object shown in an image, many models designed to perform other tasks need to internally learn to reconstruct shapes and lift up a 2D projection of a 3D real-world object.

This is, indeed, also true for humans. For example, if we are shown a picture of the front of a car, and we are asked to imagine how the image would look if we rotate the car 90° or 180°, most of the time we will be able to "imagine" how the car looks from that other view by reconstructing the 3D shape of the car, merging information inferred from the image with our experience and knowledge of kinds of car models (hatchback, sedan, convertible, etc), and then rotating 90° or 180°.

Aware of the importance of integrating 3D shape reconstruction in Computer Vision tasks in which geometric clues play a vital role, a wide sector of researchers have published works that involve it. We will briefly overview some of the most related to the task we are addressing.

2.2 Some previous work

A large amount of work has been carried out during the last decade in reconstructing 3D shape and location of objects, as well as estimate cars shape and pose in the context of autonomous driving. We will only briefly present four of them that we consider that are highly related to the task developed in this work, although some of them address very different problems.

1. *3D-Aware Scene Manipulation via Inverse Graphics* [27]. This work carried out by researchers from MIT CSAIL shows a new method for manipulating images via inverse graphics. The main novelty of this approach is that it seeks to add 3D knowledge to

the network to the disentanglement of a scene representation. To do so they proposo 3D scene de-rendering networks (3D-SDN) to incorporate disentangled representations for semantics, geometry and appearance. The 3D geometric branch basically consists in a ResNet-18 [11], and actually aims to reconstruct the geometry of a masked car (car mask has been obtained with Mask-RCNN [10]) by predicting scale, rotation and translation of the car with respect to the camera system of coordinates, and simultaneously classifying among a list of eight CAD models and predicting a set of FFD coefficients [21].

2. *Reconstructing Vechicles from a Single Image: Shape Priors for Road Scene Understanding* [14]. This is an approach for monocular 3D reconstruction of cars in the context of autonomous driving. The fundamental idea of the paper is that estimating shape and pose simultaneously from monocular information is an ill-posed problem. Thus they propose an approach that aims to decouple pose and shape estimation problems. To do so, they demonstrate that prior knowledge about how 3D shapes of cars are projected onto an image can be used to reason about how to "back-project" from 2D to 3D. This is achieved by means of a *shape-aware adjustment* scheme that, given a shape prior and keypoint detections of a car, estimates its 3D pose and shape, even if some parts of the car are occluded.

3. *3D-R2N2* [4]. 3D-R2N2 stands for *3D Recurrent Reconstruction Neural Network*, and is 3D reconstruction framework that unified single and multi-view problems. They introduced a novel architecture called 3D Convolutional LSTM as an implementation of the hidden states of 3D RNN. Their network was made up of three modules: a 2D Convolutional Neural Network (2D-CNN), a *3D Convolutional Long Short-Term Memory Unit* (3D-LSTM) and a *3D Deconvolutional Neural Network* (3D-DCNN). The 2D-CNN encoded image features from the image, while the 3D RNN allowed the network to retain what it has seen and to update the memory after seeing new images. The 3D-DCNN outputed a voxel-wise probability of occupancy in a 3D grid.

4. *Pixel2Mesh* [26]. See section 2.3.

2.3 Pixel2Mesh

Pixel2Mesh [26] is a powerful end-to-end deep learning architecture designed to produce a 3D triangular mesh of an object given a single RGB image. The method is based on the gradual deformation of an ellipsoid mesh using a graph-based convolutional network, which is affected by features extracted from the RGB image. The predicted mesh is refined using a coarse-to-fine fashion, after putting the mesh through different stages, with increasing number of vertices. Due to the importance of this network in our work, we will go into in detail in this whole section.

2.3.1 Graph-based convolutional networks

A 3D mesh can be described as an undirected graph, that is, as a collection of vertices V and edges E. The set of edges itself defines a relation between vertices, such that a pair of vertices are related if they constitute the endings of an edge: $u \sim v$ if and only if $uv \in E$.

To such collection we can also add a set of features that are attached to each vertex, $F = \{f_v\}_{v \in V}$. In order to mold these graph features, the authors of Pixel2Mesh choose to use graph-based convolutional network (GCN). These features are modified by a GCN according to the relations defined by the graph, in the same way that features of a regular CNN are modified according to the adjacency defined by pixels in an image. We briefly overview graph convolutions here below. However, for a more detailed explanation see [1].

Specifically, the new set of features $F' = \{f'_v\}_{v \in V}$ after a graph convolutional layer is

$$f'_v = w_0 f_v + \sum_{u \sim v} w_1 f_u. \tag{2.1}$$

If the original features have d channels and we want to convolve them to features of d' channels, then each of the weights w_0, w_1 must have $d \times d'$ parameters.

Let us show how this operation can be expressed in a vectorized way by means of the affinity matrix. The affinity matrix is a $n \times n$ matrix (where $n = |V|$ is the number of vertices) such that

$$A_{uv} = \begin{cases} 1 & \text{if } u \sim v \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

That is, two vertices are related by an edge if and only if the corresponding affinity matrix entry equals 1. It is straight-forward to see that equation (2.1) can be written as

$$f'_v = w_0 f_v + \sum_{u \in V} A_{vu} w_1 f_u \tag{2.3}$$

If a $\boldsymbol{F} = (f_u)$ and $\boldsymbol{F}' = (f_v)$ are vectors containing features before and after the graph convolution layer, the we can write

$$\boldsymbol{F}' = w_0 \boldsymbol{F} + w_1 A \boldsymbol{F} \tag{2.4}$$

What makes this last equation relevant is that, as opposed to equation (2.1), it allows us to write all operations in a fully vectorized way, which is manifestly positive for a PyTorch implementation.

Observe that the products by w_0 and w_1 act as fully-connected layers with d input feature channels and d' output feature channels. Consequently, graph-based convolutional layer weights can be learned as those that constitute fully-connected layers.

2.3.2 Model components

Pixel2Mesh constitutes an end-to-end deep-learning framework that progressively deforms and *upsamples* a 3D ellipsoid mesh to match the input image shape, assuming the knowl-

edge of the intrinsics camera matrix. To perform such task the network is made up of two components: an *image feature network* and a *cascaded mesh deformation network*.

On the one hand, the image feature network is merely a VGG-16 [22], an extensively used architecture for image feature extraction. On the other hand, the cascaded mesh deformation network is comprised of three deformation blocks, where each block consists of a *perceptual feature layer* and a *graph convolutional network*. Between each pair of consecutive deformation blocks a *graph unpooling layer* is placed to "upsample" the previous output to a higher resolution. That way, each deformation block works with meshes of increasing resolution, being the input to each block the upsampled output of the previous block (and hence, it is a clear example of a *cascaded* network).

Let us look at each of this components in some detail.

Image feature network. Due to the extended popularity and effectiveness of this network for image feature extraction tasks, the authors of Pixel2Mesh decide to use a VGG-16 [22] as the image feature network. Specifically, features extracted from layers `conv3_3`, `conv4_3` and `conv5_3` are concatenated to form a *perceptual feature* $\boldsymbol{P} \in \mathbb{R}^{w \times h \times d}$, of dimension $d = 1280$.

Perceptual feature layer. The function of the perceptual feature layer is to adapt properly image features to the mesh topology. This is achieved by assuming knowledge of the intrinsics camera matrix K.

Given a vertex v of the input mesh we project its coordinates p_v it onto the left camera plane, obtaining in this process a point $\bar{p}_v = (x, y) \in \mathbb{P}$ belonging to the projective camera plane and satisfying $\bar{p}_v \sim K p_v$. Then, the perceptual feature of a given vertex v is the feature corresponding to its projection onto the camera plane. Since the image plane is actually a discrete set of points, $[w] \times [h]$, features are pooled from the original left and right feature maps using bilinear interpolation, i.e., if $x = \lfloor x \rfloor + \delta x$, $y = \lfloor y \rfloor + \delta y$, the final feature is the weighted sum of

$$\begin{array}{lll} P(\lfloor x \rfloor, \lfloor y \rfloor) & \text{with weight} & (1-\delta x)(1-\delta y) \\ P(\lfloor x \rfloor, \lceil y \rceil) & \text{with weight} & (1-\delta x)\delta y \\ P(\lceil x \rceil, \lfloor y \rfloor) & \text{with weight} & \delta x(1-\delta y) \\ P(\lceil x \rceil, \lceil y \rceil) & \text{with weight} & \delta x \delta y \end{array} \tag{2.5}$$

that is

$$\begin{aligned} f_v = (1-\delta x)(1-\delta y)P(\lfloor x \rfloor, \lfloor y \rfloor) + (1-\delta x)\delta y P(\lfloor x \rfloor, \lceil y \rceil) + \delta x(1-\delta y)P(\lceil x \rceil, \lfloor y \rfloor) \\ + \delta x \delta y P(\lceil x \rceil, \lceil y \rceil), \quad \text{where } (x, y) = \bar{p}_v \end{aligned} \tag{2.6}$$

Using bilinear interpolation not only provides a meaningful role to the non-integer part of each camera projective plane point, but ensures that this process is fully differentiable and does not stop gradient back-propagation during training.

As a result of the previous operation, we obtain a perceptual feature f_v of dimension 1280 for all vertices $v \in V$. This feature is concatenated with the 128-dim shape feature g of

the input mesh (in the case of the first block, there is no shape feature so vertex coordinates of the ellipsoid are used as shape feature), to form a 1408-dim feature $g \otimes f \in \mathbb{R}^{n \times 1408}$ (or 1283-dim feature in the case of the first block), that constitutes the input for the graph convolutional network.

Graph convolutional network. The 1408-dim feature (or 1283 in the case of the first block) encoding 3D-shape and 2D image information serves as input of a graph convolutional network (GCN), that uses this feature to predict a new location for the vertices. The authors of Pixel2Mesh design a very deep GCN with shortcut connections. It is comprised of 14 graph residual convolutional layers with 128 channels each, and they refer to it as G-ResNet.

The reason they opt for a very deep structure is that the graph-convolutional layer defined before has a small receptive field (only immediate neighbours of a vertex exchange information) and, hence, it is necessary a large number of convolutions to allow the exchange of information between distant vertices. This problem is also addressed stretching the network into different stages, starting with a low number of vertices to allow information exchange between distant vertices.

Graph unpooling layer. Since the goal is to build a cascaded network in which every stage has increasing resolution, we need to increase the number of vertices after each deformation block. To do so, the authors of Pixel2Mesh build a layer adds a vertex at the centre of every edge of the graph (see figure 2.1). This new vertex coordinates as well as its 3D features are an average between its neighbours. Moreover, if three of these vertices are added to the same triangle, they connect them, creating in the process 4 triangles from one triangle of the previous mesh. Observe that this process preserves the degrees of each vertex, so the graph remains balanced after going through this layer.

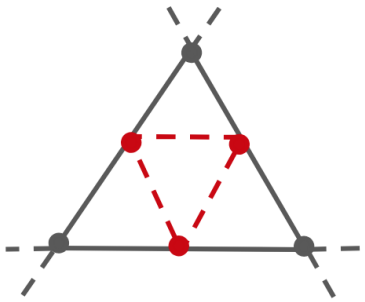


Figure 2.1: Unpooling layer adds a vertex at the centre of every edge of the graph, and connects three of these vertices if they are added to the same triangle.

The overall architecture is shown in figure 2.2

2.3.3 Losses

During training time, the network is supervised by four kinds of losses. These losses are responsible for regressing the output vertices to the ground truth shape, but also of ensuring that vertices converge to a smooth and uniform shape. Two of the four terms account for direct comparison with ground truth vertices and normals, whereas the other two are responsible for adding some regularization that prevents the network of getting stuck into some local minima.

(a) **Chamfer loss**. *Chamfer distance* (CD) [5] is used as a penalty that regresses the pre-

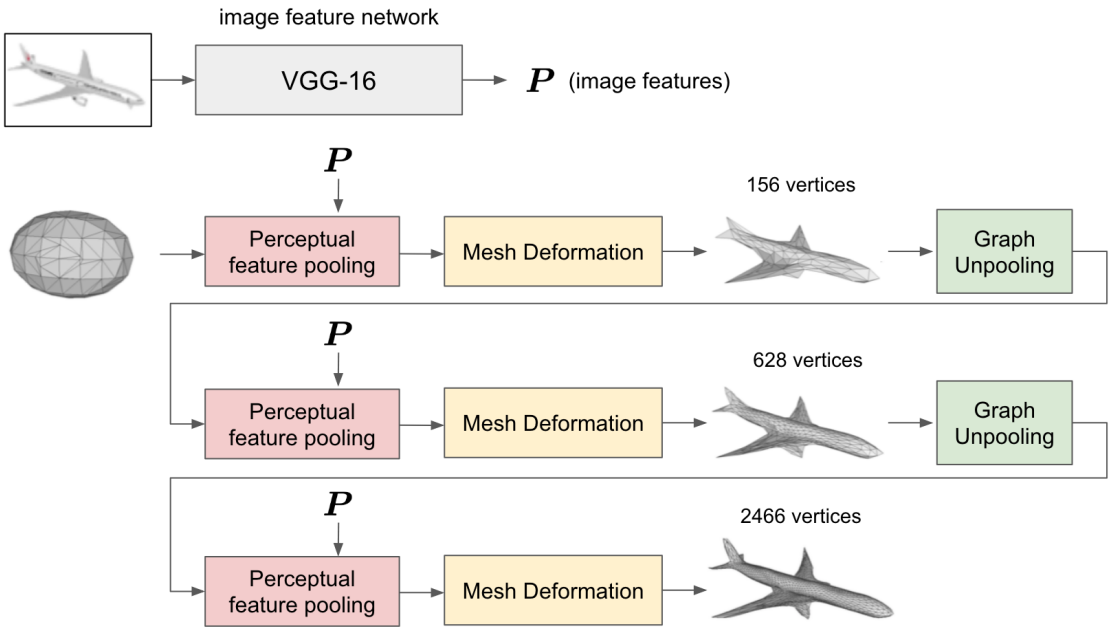


Figure 2.2: Overall architecture of Pixel2Mesh.

dicted point cloud to the ground truth. It measures the distance between ground truth points and predicted mesh vertices:

$$L_{\text{chamfer}} = \sum_{p} \min_{q} \|p - q\|_2^2 + \sum_{q} \min_{p} \|p - q\|_2^2, \tag{2.7}$$

being q ground truth mesh vertices. ls it is pointed out in [5], this function is actually not a "distance", since it doesn't fulfill triangular inequality. Observe that this term is not enough to ensure the convergence to suitable meshes, and more constraints are necessary.

(b) **Normal loss**. This term encourages the normal of a locally fitted tangent plane to be consistent with the ground truth normal. For each given predicted mesh vertex p let $q(p)$ be the closest ground truth vertex found when calculating the chamfer loss and $n_{q(p)}$ the observed surface normal from ground truth at that point, then

$$L_{\text{normal}} = \sum_{p} \sum_{qp \in E} \|(p - q)^T n_{q(p)}\|_2^2. \tag{2.8}$$

(c) **Laplacian regularization**. This term prevents the mesh from deforming too much. That way we ensure that the output mesh has a smooth surface like the input ellipsoid. It is defined as:

$$L_{\text{laplace}} = \sum_{p} \|\delta_p - \delta'_p\|_2^2 \tag{2.9}$$

where p' and p are the vertices before and after a deformation block, and δ_p is the laplace coordinate of a point p, which is defined as

$$\delta_p = p - \sum_{q \sim p} \frac{q}{|\{q : q \sim p\}|}. \tag{2.10}$$

In other words, the laplace coordinate of a point is obtained by substracting to it the average of its neighbours.

(d) **Edge length regularization**. To prevent the mesh from having flying vertices, we penalize long edges by adding the following term:

$$L_{\text{edge}} = \sum_{p} \sum_{qp \in E} \|q - p\|_2^2. \tag{2.11}$$

The overall loss is a weighted sum of all these terms: $L = \lambda_c L_{\text{chamfer}} + \lambda_n L_{\text{normal}} + \lambda_l L_{\text{laplace}} + \lambda_e L_{\text{edge}}$.

Chapter 3

3D mesh prediction for cars

3.1 Introduction

During the last years, Artificial Intelligence has become a cornerstone of the development of Autonomous Driving. Building a smart and safe convivence of all kinds of vehicles in the road certainly determines a huge list of obstacles to deal with. The spectrum of challenges is so rough that it widely covers most of the research areas that make up Machine Learning (ML). This spectrum is essentially split into two major sets of tasks: those that focus on interpreting, recognizing and reconstructing the real world around a car during its circulation; and those that deal with the policy-making processes conditioned to a certain interpreted sensor information. The last is usually addressed by Reinforcement Learning (RL) techniques, whereas the first is closely tied to Deep Learning (DL). We cannot afford to ignore the fact that having a good reconstruction and processing of the sensor information is crucial to have at one's disposal as much information as needed in order to enhance any right decision-making algorithm.

For that reason, several efforts have been allocated to assemble and annotate large collections of samples involving recognition and reconstruction of the 3D world captured by car sensors. One of the most popular and well-developed datasets is KITTI [7] [17]. This project of Karlsruhe Institute of Technology and Toyota Technological Institute at Chicago holds a wide variety of benchmarks that include stereo, optical flow, visual odometry, 3D object detection, semantic segmentation, etc, based on datasets recorded from a wagon equipped with high-resolution cameras, a LIDAR sensor, and GPS, that drives around rural areas and highways of the mid-size city of Karlsruhe [7]. A smaller subset of this data has been processed and manually annotated with 2D and 3D bounding boxes, segmentation masks, etc.

Due to the fact that manually annotating images is a slow, tedious and complicated process, it is very expensive to really be able to fill all the data labels. For that reason, only a small portion of all KITTI raw data constitutes the benchmarks. The number of samples for each benchmark oscillates between 300 or 7K samples, whereas the raw data adds up to more than 100K samples. In order to boost and reduce the cost of annotating data samples,

researchers are focusing too on developing computer assistants and software that facilitates the annotation task. We are speaking of tools that roughly complete a large percentage of the job, leaving a pretty good starting point for the annotator.

The idea behind our project is that 3D reconstruction can bring solid starting points for point cloud annotation. Focusing only on cars, a model trained for reconstructing cars from a "KITTI-like" image might be able to provide the annotator with a pretty good first approximation of the point cloud or 3D mesh, which then would be only followed by few manual tweaks by the annotator. Moreover, the approximation provided by this model might be good enough to already serve in most tasks demanding a precise point cloud for vehicles.

With the purpose of building such a model, we adapt Pixel2Mesh [26] to perform of 3D mesh reconstruction for cars with the support of a 3D box, and we render a very large synthetic dataset of cars that we use to train and test our model.

3.2 Rendered data

For the task of 3D mesh prediction for cars we have rendered a synthetic dataset of white-background car images in road-like scenes. It is based on 3K car models obtained from ShapeNetCore [3] (see figure 3.1). Even though these cars represent accurate replicas of real-world existing cars, they do not have real-world dimensions. To overcome this problem we have annotated each car model with estimations of their real-world counterpart car dimensions of width, heigh and depth.

Figure 3.1: Some of the car models we use to render our dataset. They all have been obtained from ShapeNetCore dataset [3].

To synthesize these scenes we use OpenGL 4.0/C++. Specifically, we use the C++ library GLEW.

3.2.1 3D scene data

For each car model we render 100 scenes with the car centre located at $[X, Y, Z]$, where X, Y and Z are random variables defined as

$$\begin{aligned} X &= x_0 + \Delta x \cdot U_x \\ Y &= y_0 + \Delta y \cdot U_y \\ Z &= z_0 + \Delta z \cdot U_z. \end{aligned} \tag{3.1}$$

Here U_x, U_y, U_z follow uniform distributions $U(-1, 1)$. Moreover, we allow the car to uniformly rotate along the Y axis and we also allow some noisy slope along the X direction (horizontal direction parallel to the camera plane). This is achieved using two random variables describing angles of rotation along X and Y axis: $\alpha_x \sim U(-a, a)$, with $a \ll \pi$ being a small slope, $\alpha_y \sim U(-\pi, \pi)$.

By construction, $\boldsymbol{t} = (X, Y, Z)$ is a random vector describing the translation of the bounding box. The random variable α_x defines a random rotation matrix along the X axis:

$$R_x(\alpha_x) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha_x & -\sin\alpha_x \\ 0 & \sin\alpha_x & \cos\alpha_x \end{bmatrix} \tag{3.2}$$

and, similarly, α_y defines a random rotation matrix along the Y axis:

$$R_y(\alpha_y) = \begin{bmatrix} \cos\alpha_y & 0 & \sin\alpha_y \\ 0 & 1 & 0 \\ -\sin\alpha_y & 0 & \cos\alpha_y \end{bmatrix}. \tag{3.3}$$

Putting it all together, we can fill the entries of a matrix $T \in \mathbb{R}^{4\times4}$ describing the rigid transformation of the 3D bounding box in homogeneous coordinates:

$$T = \begin{bmatrix} 1 & 0 & 0 & X \\ 0 & 1 & 0 & Y \\ 0 & 0 & 1 & Z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} & R_x(\alpha_x) & & 0 \\ & & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} & R_y(\alpha_y) & & 0 \\ & & & 0 \\ & & & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.4}$$

This is actually the way we store rigid transformations of each scene in our dataset.

As we said before, we have manually annotated cars with their real-world dimensions of width w_x, height w_y and depth w_z. We store this information as a matrix $S \in \mathbb{R}^{4\times4}$ (scale matrix), that transforms a normalized point cloud of car of unit width, height and depth into the real-world dimensions car point cloud:

$$S = \begin{bmatrix} w_x & 0 & 0 & 0 \\ 0 & w_y & 0 & 0 \\ 0 & 0 & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.5}$$

Hence, given a normalized point cloud of the car P, each point $p = [x, y, z] \in P$ will be located for rendering at $[x', y', z']$, where

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = TS \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}. \tag{3.6}$$

Coordinates of each point of the normalized point cloud are stored. Moreover, we compute the observed normals at each vertex of the shape. To be precise, since ShapeNet models are triangular 3D meshes, we can compute face normals by computing the cross product $\boldsymbol{n}_f = \frac{\boldsymbol{e}_1 \times \boldsymbol{e}_2}{\|\boldsymbol{e}_1 \times \boldsymbol{e}_2\|}$ of a pair of edges $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$ of the triangle (with proper orientation). Then, to each vertex we assign as observed normal the normalized sum of all face normals of faces that contain that vertex, that is, if F_v is the set of faces containing a vertex v, the observed normal for vertex v is:

$$\boldsymbol{n}_v = \frac{\sum_{f \in F_v} \boldsymbol{n}_f}{\| \sum_{f \in F_v} \boldsymbol{n}_f \|}. \tag{3.7}$$

Hence, we possess two sets $P = \{p_v\} \in \mathbb{R}^3$ and $N = \{\boldsymbol{n}_v\}$ of, respectively, point coordinates and normals for any mesh describing a ShapeNet normalized model. It is important to notice that normals can also be transformed and rescaled using T and S, but using vector homogeneous coordinates, and we will cover this in detail in section 3.3.1.

See figure 3.2 for an example of a ground truth point cloud sampling and its corresponding 3D bounding box. The size of the point cloud may oscillate between 10K and 80K vertices, depending on the model and how precise the mesh replica is. Moreover, points are not necessary uniformly distributed, but can be more concentrated on certain spots of the car that require more detail.

3.2.2 2D projected data

The 3D car normalized mesh is scaled and transformed according to S and T, and faces are filled with the corresponding color shaders. Color and z coordinate are projected onto a camera plane to render RGB images and depth maps respectively.

The camera plane is located at the origin and it is orthogonal to the Z plane. However, since image pixel axes usually increase to the right and down, we invert axes direction. To be precise the camera plane has axes (x, y), with $x = -X$ and $y = -Y$, where (X, Y, Z) are the 3D axes. This is taken into account by the camera intrinsic matrix K, that reveals negative focal lengths $f_x = 309.0193 < 0$ and $f_y = 309.0193 < 0$. We set a resolution of 768×256 pixels. Therefore, the principal point offset is $(p_x, p_y) = (334, 128)$. Putting it all together, the camera intrinsics matrix looks like

$$\begin{bmatrix} f_x & 0 & p_x \\ 0 & f_y & p_y \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -309.0193 & 0 & 384 \\ 0 & -309.0193 & 128 \\ 0 & 0 & 1 \end{bmatrix}. \tag{3.8}$$

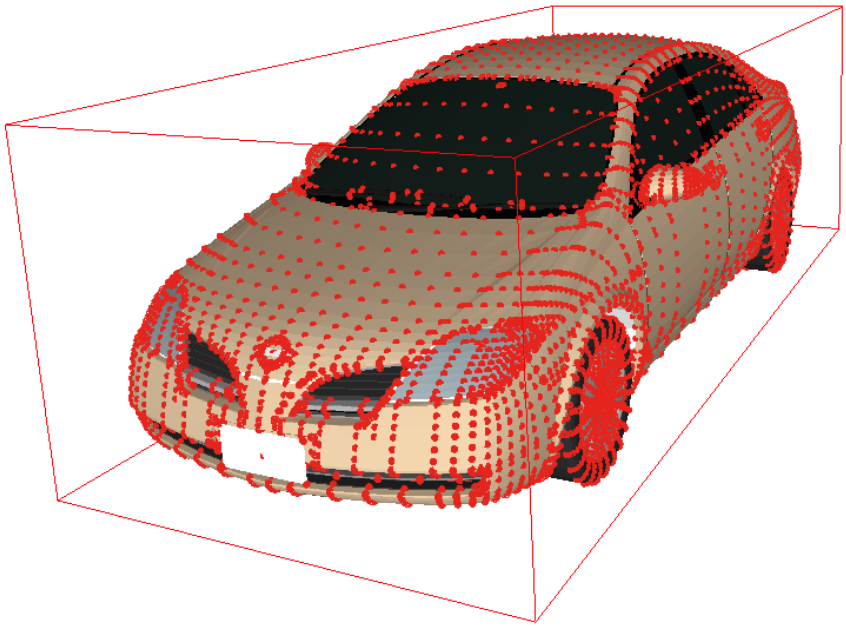


Figure 3.2: Ground truth point cloud and 3D bounding box of a 2003 Nissan Primera.

3.3 Predicting car meshes given 3D bounding box

In this section we are going to discuss a more constrained problem: instead of predicting 3D mesh absolute coordinates for a car given only a masked RGB image of it (in a road-like scenario), we are going to assume that we also know a precise ground truth 3D bounding box.

3.3.1 3D bounding box normalization

By 3D bounding box we mean a 3D box centered at the car mean point, with the same orientation that the car, so that car reference system of coordinates is aligned with the box and the dimensions of the box match those of the car. This is equivalent to giving three parameters for the dimensions of the box (w_x, w_y and w_z), and a rigid body transformation $T \in SE(3, \mathbb{R})$ that maps a box of these dimensions centered at the origin to the actual position and orientation of the car.

By normalizing the ground truth point clouds to that bounding box, we can considerably reduce the space of possible outcomes expected from our graph convolutional network. Specifically, let $(P_{gt}, N_{gt}) \subset \mathbb{R}^3 \times \mathbb{R}^3$ be our ground truth set of absolute vertex coordinates and normals observed at those points. We define the normalized point/normal cloud as follows: we use homogeneous coordinates $[x, y, z, 1] \in \mathbb{R}^4$ to express points $p \in P_{gt}$, and infinity points for normals $[n_x, n_y, n_z, 0] \in \mathbb{N}_{gt}$, since they are vectors.

Given the 3D bounding box transformation, which must be a proper affine movement $T \in \mathbb{R}^{4\times4}$ (that is, a translation and/or rotation of the space), we can normalize both normals

N_gt and points P_{gt} by applying T^{-1} to all their elements:

$$\bar{P}_{gt} = T^{-1}P_{gt}, \quad \bar{N}_{gt} = T^{-1}N_{gt} \tag{3.9}$$

This already means a remarkable reduction of the space of possible outcomes, since now all car point clouds are centered at the origin and have fixed orientation.

We can further constrain the space of possible coordinates for the predicted vertices by normalizing according to the dimensions of the 3D bounding box. Specifically, let us define a scaling matrix $S = \text{diag}(w_x, w_y, w_z, 1)$, where w_x, w_y and w_z are the 3D box dimensions, meaning that

$$-w_x/2 \leq x \leq w_x/2, \quad -w_y/2 \leq y \leq w_y/2, \quad -w_z/2 \leq z \leq w_z/2, \tag{3.10}$$

for any $[x, y, z] \in \bar{P}_{gt}$. In other words, $w_x = \max x - \min x$, $w_y = \max y - \min y$ and $w_z = \max z - \min z$.

We can redefine our normalized set of points as $\bar{P}_{gt} := S^{-1}\bar{P}_{gt}$ and its corresponding set of normals $\bar{N}_{gt} := \texttt{normalize}(S^{-1}\bar{N}_{gt})$, where

$$\begin{aligned} S^{-1}\bar{P}_{gt} &= \{S^{-1}[x, y, z, 1]^T \,:\, [x, y, z]^T \in \bar{P}_{gt}\} \\ \texttt{normalize}(S^{-1}\bar{N}_{gt}) &= \left\{ \frac{S^{-1}[n_x, n_y, n_z, 0]^T}{\|S^{-1}[n_x, n_y, n_z, 0]^T\|_2} \,:\, [n_x, n_y, n_z]^T \in \bar{N}_{gt} \right\} \end{aligned} \tag{3.11}$$

Clearly, after this scale-normalization all dimensions will be bounded in magnitude by $1/2$, that is,

$$-1/2 \leq x \leq 1/2, \quad -1/2 \leq y \leq 1/2, \quad -1/2 \leq z \leq 1/2. \tag{3.12}$$

So, as promised, this leads to a significant reduction of the range of expected reasonable values for predicted coordinates.

Besides considerably reducing the dimensionality of the problem, this approach also allows us to exploit the benefits of decoupling mesh estimation into pose estimation plus shape estimation. [6] exhaustively examines 3D reconstruction from a single image and actually proves that estimating shape and pose simultaneously from a single image is an ill-posed problem that suffers from several ambiguities. Hence, a two-phase mesh estimation from a single image consisting of 3D bounding box prediction followed by shape prediction should lead to better results than a joint model.

3.4 Architecture

In this section we are going to describe the architecture of an adaptation of Pixel2Mesh to address our problem. Let us stand back for a moment and describe what is the task we want our network to carry out. On the one hand, the input of the network will be:

(a) A white-background RGB image of a car, as the example showed in section 3.2, with a bounding box of the car.

(b) The camera intrinsics K of the camera that has taken the picture.

(c) The 3D bounding box of the car, parametrized by: a matrix $T \in \mathbb{R}^{4\times4}$, representing the orientation and translation of the bounding box with respect to the reference system of coordinates; a matrix $S \in \mathbb{R}^{4\times4}$ that transforms a normalized car to match the bounding box dimensions.

On the other hand, as output of the network we want to predict the 3D location of all vertices of our mesh graph in a normalized bounding box system of coordinates, so that after unnormalizing these coordinates they are as close as possible to the ground truth point cloud.

3.4.1 Image feature network

As Pixel2Mesh, our network consists of two components: an *image feature network* and a *cascaded mesh deformation network*, where the deformation network pools features from the image feature network to produce shape features. We choose as well a VGG-16 [22] for the image feature network. However, in this case we have a different type of input. Instead of rendering images of a certain car from several points of view, with a fixed distance (as Pixel2Mesh), we have a fixed camera, located at a heigh approximately equivalent to a typical car on-board camera set up, which is around 1.5 meters over the road, and rendered cars are located at random positions over the road plane as detailed in section 3.2.

Moreover, we will have typically a small car compared to image dimensions. This means that most of the RGB scene is empty and is useless for the image feature network. For that reason we crop the image to the 2D bounding box that we consider as input. By doing this we also substantially reduce the spatial dimensions and, hence, the amount of memory occupied by feature maps extracted by the VGG. We also pad the cropped image (10% each side) as well as resize it to match VGG input spatial dimensions (224×224). To sum it up, this is the whole process: if $w \times h$ are the spatial dimensions of the input image and $b_x \times b_y$ the 2D bounding box dimensions, then

$$I \in \mathbb{R}^{w\times h\times 3} \xrightarrow{\text{crop}} I \in \mathbb{R}^{b_x\times b_y\times 3} \xrightarrow{\text{pad } 10\%} I \in \mathbb{R}^{1.2b_x\times 1.2b_y\times 3} \xrightarrow{\text{resize}} I \in \mathbb{R}^{224\times224\times3} \tag{3.13}$$

See figure 3.3 to see an example of input of the VGG.

Please note that this process alters the camera matrix. Thus we must modify camera intrinsics to take this process into account, in such a way that we can pool features from the image later. Specifically, if (c_x, c_y) is the pixel coordinate of the left-bottom vertex of the 2D bounding box after padding with a 10% each side, and $(1.2b_x, 1.2b_y)$ are the dimensions of the padded bounding box, then the previous process is equivalent (in terms of pixel coordinates) to translating all pixels by $(-c_x, -c_y)$ and then rescaling by $224/1.2b_x$ and $224/1.2b_y$ axes x and y. So, putting it all together, this can be described by a concatenation of translation and a rescaling:

$$N = \begin{bmatrix} \frac{224}{1.2b_x} & 0 & 0 \\ 0 & \frac{224}{1.2b_y} & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -c_x \\ 0 & 1 & -c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{3.14}$$

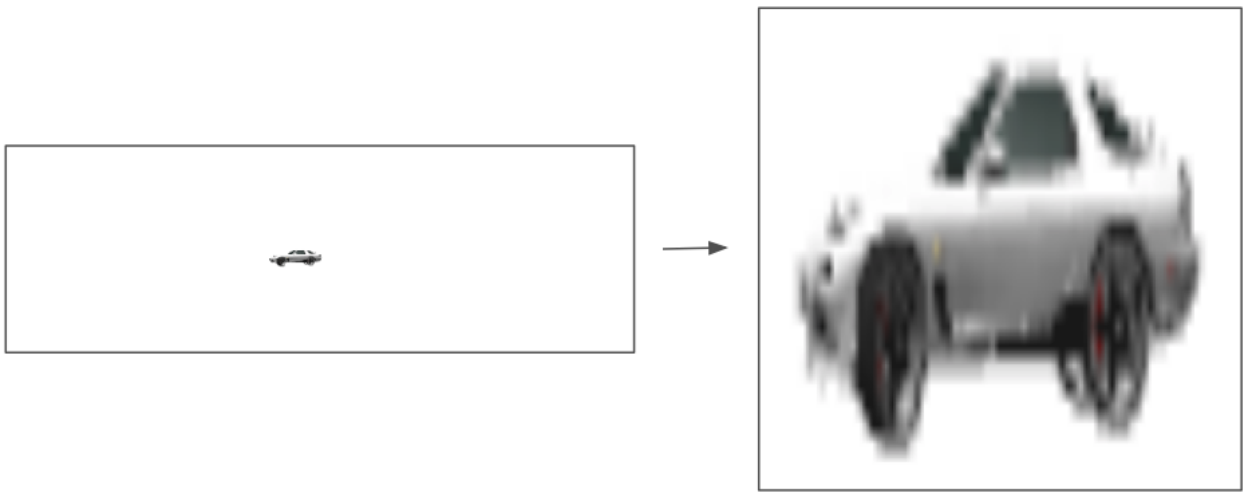


Figure 3.3: Example of transformation of an image to be fed into the VGG.

or

$$N = \begin{bmatrix} \frac{224}{1.2b_x} & 0 & \frac{224}{1.2b_x}c_x \\ 0 & \frac{224}{1.2b_y} & \frac{224}{1.2b_y}c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{3.15}$$

As expected, this change of coordinates can be expressed by a linear transformation. Multiplying homogenous pixel coordinates of the original image by this matrix maps them into the 224×224 new system of pixel coordinates. Therefore, $K' = NK$ is the right camera matrix for describing the 2D projection of 3D vertices onto the system of pixel coordinates used as input of the image feature network.

Once our original image is properly transformed to be fed into the image feature network, which consists in the feature extraction core of the VGG-16 [22] (first five convolutional blocks), we take features extracted from layers `conv3_3`, `conv4_3` and `conv5_3` and concatenate them to form a perceptual feature $\boldsymbol{P} \in \mathbb{R}^{w \times h \times d}$, of dimension $d = 1280$.

3.4.2 Mesh deformation network

Following Pixel2Mesh approach, we provide the network with a cascaded mesh deformation network, consisting of three mesh deformation blocks of increasing number of vertices. The input to the network is an ellipsoid and the end of each block we pool features from the image and "interpolate" the mesh using an unpooling layer in order to increase the resolution for the next deformation block.

We already pointed out previously how in our problem we render images of cars at random positions and poses with a fixed camera, instead of a fixed object from which we take several views. This means that in principle we will not have zero-centered coordinates as in Pixel2Mesh, where all models are centered at the origin, but point clouds located at random positions of the space of coordinates (above the ground plane). However, we already discussed in 3.3.1 how 3D bounding boxes can be used to normalize our point clouds. This

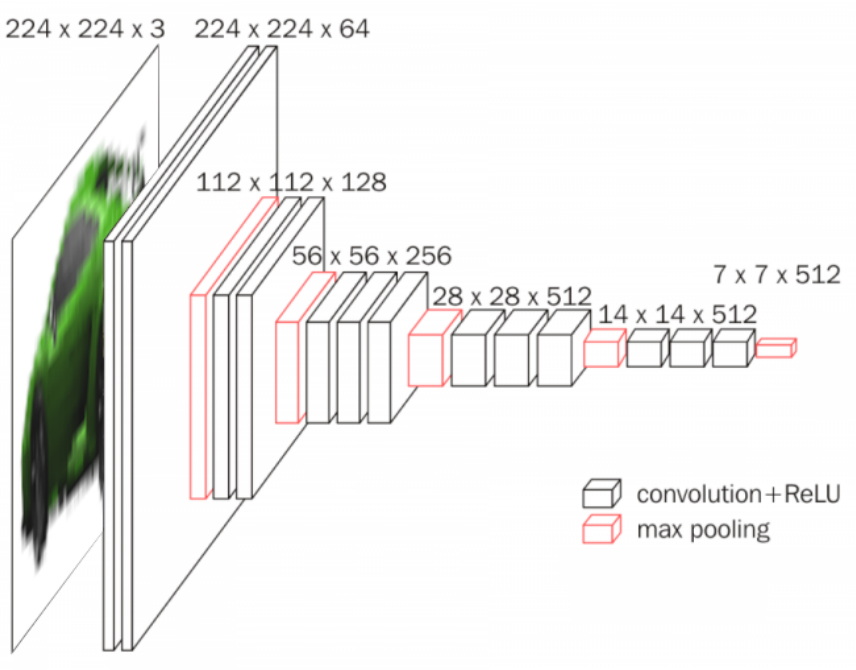


Figure 3.4: Image feature network. It consists of the first five blocks of the VGG-16 [22], to which we feed the cropped and rescaled image of the car.

normalization basically constraints our space of possible outputs to the inside of the 3D bounding box, in such a way that our network only has to focus on predicting the right shape.

Nevertheless, this also makes a difference with respect to Pixel2Mesh approach, in which there is a fixed center, but unconstrained dimensions and rotation of the object. In our case, we have a fixed orientation of our mesh and also fixed dimensions, so that our predictions lie inside a 3D cube centered at the origin:

$$-1/2 \leq x \leq 1/2, \quad -1/2 \leq y \leq 1/2, \quad -1/2 \leq z \leq 1/2. \tag{3.16}$$

Since our normalized point cloud $\bar{P}$ is now obtained from the original point cloud P as $S^{-1}T^{-1}P$, we also have to modify the way we project vertices onto the camera plane to pool features. Specifically, let V be our current set of vertices of our mesh onto which we want to project image features. For each vertex $v \in V$ we have a 3D normalized coordinate $\bar{\boldsymbol{p}}_v \in \mathbb{R}^3$. This coordinate is written in the coordinate system of the 3D bounding box. Thus, in order to project this coordinate onto the camera plane we have to first transform this coordinate into the world coordinate system, and this is done by scaling and then translating and rotating the vertices adequately, i.e., the coordinate we project is actually

$$\begin{bmatrix} \boldsymbol{p}_v \\ 1 \end{bmatrix} = TS \begin{bmatrix} \bar{\boldsymbol{p}}_v \\ 1 \end{bmatrix}, \tag{3.17}$$

in terms of homogenous coordinates. After this transformation we can now project onto the 2D plane using the camera matrix K' described before. Putting it all together, it could be

all summed up into a single step by considering $NKTS$ as the "effective" camera matrix. We obtain in that case the homogeneous coordinates of the 2D projected vertex in one single step:

$$\text{2D projection of } \bar{\boldsymbol{p}}_v = NKTS \begin{bmatrix} \bar{\boldsymbol{p}}_v \\ 1 \end{bmatrix} = K_{\text{eff}} \begin{bmatrix} \bar{\boldsymbol{p}}_v \\ 1 \end{bmatrix} \tag{3.18}$$

where $K_{\text{eff}} = NKTS$. This 2D pixel coordinate is used to pool a feature from the closest pixels of the feature map via bilinear interpolation, as in Pixel2Mesh.

The input to the network in this case is a sphere of diameter 1.25 meters centered at the origin. As opposed to what happens in Pixel2Mesh, we transform this sphere according to S and T before projecting those vertices onto the camera plane to pool features, and this constitutes an important difference between both approaches. We know that this is translated into a camera matrix K_{eff} constantly changing.

See figure 3.5 for a diagram of the overall architecture.

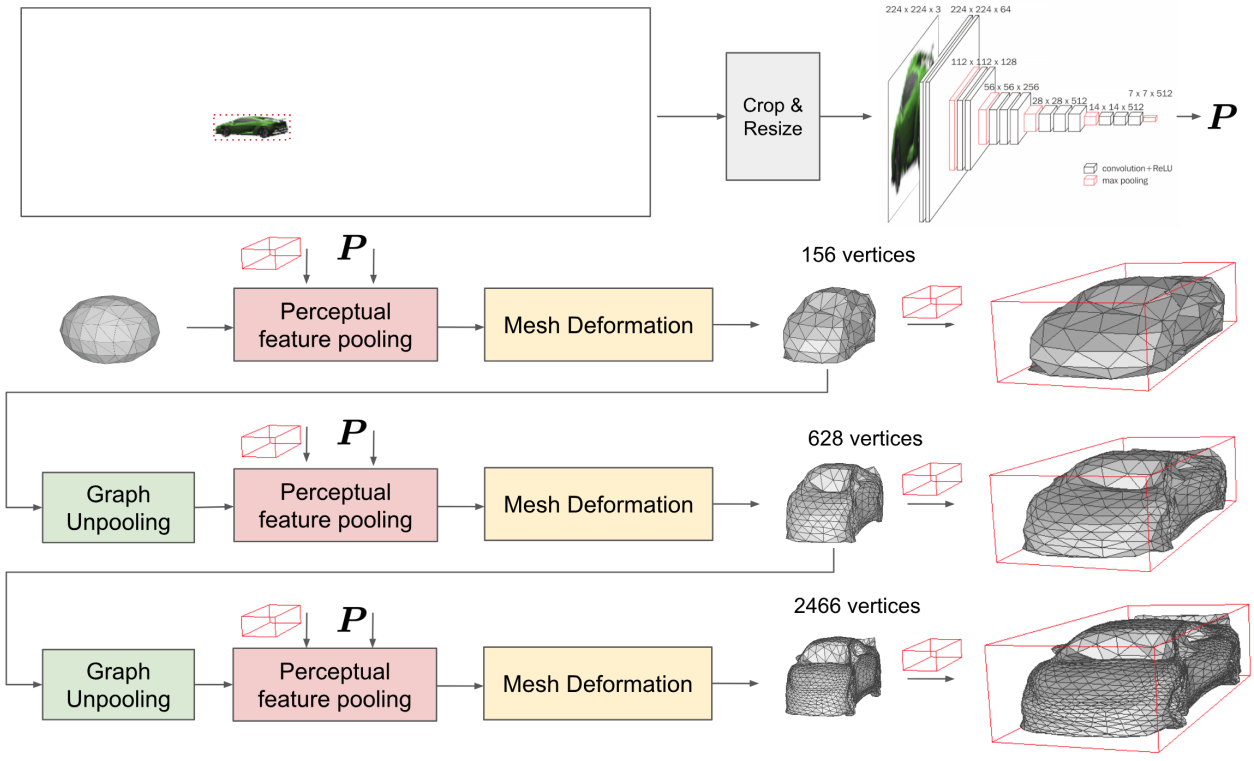


Figure 3.5: Overall architecture of the proposed model.

3.5 Losses

We adopt all losses used in Pixel2Mesh, namely the Chamfer loss, normal loss, laplacian regularization and edge regularization. We add an extra term called *move loss* between blocks (as Laplace loss) that prevents vertex coordinates to move too far away from where

they were in the previous block output. We found that without this term the optimization would not converge to a reasonable minimum, so we were forced to introduce this term to avoid meshes to twist around themselves and encourage, thus, a convex shape. Actually, this term figures in the published Pixel2Mesh implementation[1] already introduced this term between 2nd block input and output, as well as 3rd block input and output, but we found necessary to also add this term to the first block.

Moreover, we also noticed that the official implementation of the losses replaces sums by means. We decide to follow this definition instead. The final set of losses looks like:

$$\begin{aligned}
L_{\text{chamfer}} &= \frac{1}{|V|}\sum_{p}\min_{q}\|p-q\|_2^2 + \frac{\lambda}{|V_{gt}|}\sum_{q}\min_{p}\|p-q\|_2^2,\\
L_{\text{normal}} &= \frac{1}{|E|}\sum_{p}\sum_{qp\in E}\|(p-q)^T n_{q(p)}\|_2^2,\\
L_{\text{laplace}} &= \frac{1}{|V|}\sum_{p}\|\delta_p-\delta'_p\|_2^2,\\
L_{\text{edge}} &= \frac{1}{|E|}\sum_{p}\sum_{qp\in E}\|q-p\|_2^2,\\
L_{\text{move}} &= \frac{1}{|V|}\sum_{p}\|p-p'\|_2^2.
\end{aligned} \tag{3.19}$$

where p' and p are the coordinates of a vertex before and after a deformation block, and we set $\lambda = 0.55$. The overall loss is then weighted sum of all these terms: $L = \lambda_c L_{\text{chamfer}} + \lambda_n L_{\text{normal}} + \lambda_l L_{\text{laplace}} + \lambda_e L_{\text{edge}} + \lambda_m L_{\text{move}}$.

3.6 Training

The final dataset contains about 300K samples, each of which possesses 3D data (point cloud and 3D bounding box), and 2D data (RGB image, 2D bounding box, camera matrix, etc). We split this data into three subsets: *train*, *validation* and *test*, which contain, respectively, an 80%, 10% and 10% of all samples, that is, 240K, 30K and 30K images. For data augmentation, we add random hue changes to images during training. The general configuration is set to have batch-size 1, starting learning rate of 3×10^{-5}, Adam optimizer [13] with momentum constants $\alpha = 0.9$ and $\beta = 0.999$. We also add weight decay $\gamma = 5 \times 10^{-6}$ to regularize.

The initial sphere is set to have diameter 1.25 (or radius 0.625). We found this diameter worked well, since it mostly covered the whole car when it was projected onto the image. We dilate the Z axis of the normalized system of coordinates by 2, so that the proportion between axis is $1:1:2$. We found this necessary to not lose precision along the depth axis of a car, since it is usually way longer than the other two dimensions.

Even though Pixel2Mesh [26] does train jointly the whole network from the beginning, we find necessary in our case to pretrain each of the units before in order to guarantee that

[1] See https://github.com/nywang16/Pixel2Mesh for Pixel2Mesh [26] official implementation.

the model optimizes properly. The training schedule is the following: first of all, we train the first deformation block together with the image feature network for 240K iterations (1 epoch). Secondly, we fix first the weights of the first deformation block and the VGG network, and we append a second deformation block, which is then optimized for 240K iterations. Thirdly, with first and second deformation block fixed as well as the image feature network, we train the third deformation for 1 epoch. Finally, we train the whole network jointly for 2.4M iterations (10 epochs), in which we supervise the network after each of the deformation blocks.

Chapter 4

Experiments and results

4.1 Baselines and metrics

We do not have a baseline for this approach, since we are willing to train a 3D reconstruction network over a very specific kind of dataset designed for this network. However, models are similar to the ones used in Pixel2Mesh [26] since they both come from ShapeNetCore dataset [3]. The main difference is that we only work with cars and we always place cars over an imaginary road plane (to simulate a 'KITTI-like' scenario) instead of rendering random views from different angles. However, Pixel2Mesh shows results over different categories of the ShapeNet hierarchy of models, so we are going the consider their values over the category of cars as a reference of what numbers we should expect to obtain.

4.1.1 F-score

We use two different measures to quantitatively evaluate how good our model is on the test set, which are widely extended as metrics for 3D reconstruction or point cloud generation. The first measure is the F-score (%), which is the harmonic mean between precision p and recall r:

$$\text{F-score } (\%) = \frac{2pr}{p+r}. \tag{4.1}$$

Here precision and recall are measured at a certain threshold τ. On the one hand, precision is the percentage of points of the predicted mesh that can find a ground truth point at distance lower than τ. On the other, recall is the percentage of points of the ground truth point cloud that can find a point of our predicted mesh at distance lower than τ. Pixel2Mesh uses two thresholds: $\tau = 0.01$ and $\tau = 0.014$. In our case, however, we have rescaled cars of ShapeNet by a mean factor of 5.13 to have real world dimensions. In addition, Pixel2Mesh rescales ShapeNet models by a factor of 0.57. Therefore, thresholds must be multiplied by $5.13/0.57 = 9$ to recover equivalent thresholds. For that reason, we use $\tau = 9$ cm and $\tau = 18$ cm, as well as a smaller threshold, $\tau = 5$ cm.

4.1.2 Chamfer distance

The other measure we use to compute quantitatively how well our model performs on the test set is the Chamfer distance (CD). We compute it as in Pixel2Mesh:

$$\text{CD} = 1000 \times \left(\frac{1}{|V|} \sum_{p \in V} \min_{q} \|p - q\|_2^2 + \frac{1}{|V_{gt}|} \sum_{q \in V_{gt}} \min_{p} \|p - q\|_2^2 \right). \tag{4.2}$$

In this case, if we want to compare with Pixel2Mesh CD measurements, we would have to divide our values by $9^2 = 81$ (since CD has dimensions of squared unit of length).

Using this conversion, we take as a reference Pixel2Mesh mean F-score on their test set of cars, at threshold 9 cm, as our reference, which is approximately a 67%, and a Chamfer distance of approximately $0.27 \times 9^2 \simeq 22$.

4.2 Experiments

We implement our own model from scratch with PyTorch [18]. Using this framework we build the model and training setup described in chapter 3, following a training schedule that consists in four stages: three stages in which we consecutively pretrain each of the modules that make up the network for one epoch, and one last stage in which we jointly optimize the whole network. Completing the full training schedule takes 120 hours (5 days) on a Nvidia Titan Xp.

For each of these stages we set the weights detailed in table 4.1. After training every stage of the network, we validate results on the validation set and, for the full trained model, we evaluate it on the test set and measure both F-score at thresholds $\tau \in \{5, 9, 18\}$ cm and CD.

		Loss parameters [1st block]					Loss parameters [2nd and 3rd block]				
Stage	Iterations	λ_c	λ_n	λ_l	λ_e	λ_m	λ_c	λ_n	λ_l	λ_e	λ_m
1st	240K	3000	0.5	1500	300	100	3000	0.5	1500	300	100
2nd	240K	3000	0.5	1500	300	100	3000	0.5	1500	300	100
3rd	240K	3000	0.5	1500	300	100	3000	0.5	1500	300	100
4th	2.4M	3000	0.5	1500	300	0	3000	0.5	1500	300	100

Table 4.1: Weight configuration for the loss functions computed after each deformation block during each of the training stages.

4.2.1 Quantitative results

After each pretraining stage (stages 1, 2 and 3) we compute F-score at different thresholds on our validation set. We calculate for 156, 628 and 2466 vertices, unpooling if necessary.

For example, after first stage we have only trained the first deformation block (156 vertices output), so to compute F-score with 2466 vertices we interpolate twice the mesh using an unpooling layer: from 156 to 628 vertices, and from 628 to 2466 vertices. After second stage, we only have to interpolate once: from 628 to 2466 vertices.

Threshold	$\tau = 5$ cm			$\tau = 9$ cm			$\tau = 18$ cm		
Vertices	156	628	2466	156	628	2466	156	628	2466
Stage 1	4.47%	11.67%	22.73%	18.52%	38.30%	50.31%	59.91%	76.88%	80.27%
Stage 2	4.47%	17.54%	29.96%	18.52%	47.44%	58.20%	59.91%	80.45%	84.35%
Stage 3	4.47%	17.54%	35.25%	18.52%	47.44%	68.04%	59.91%	80.45%	91.94%

Table 4.2: F-score (%) on the validation set at different thresholds after pretraining for 240K iterations the first block (stage 1), second block (stage 2) and third block (stage 3). Larger is better.

We show results on validation after pretraining in tables 4.2 (F-score) and 4.3 (CD). Please notice how simple interpolation of the mesh to more vertices results in higher values of F-score and lower Chamfer distance. Also note that after third stage of training we are already obtaining more than a 67% of F-score at threshold $\tau = 9$ cm (see 4.2) and a Chamfer distance of a little bit less than 22, meaning that at least we are recovering a similar performance to Pixel2Mesh. Thus, after pretraining our model is already performing reasonably well.

	CD		
Vertices	156	628	2466
Stage 1	70.42	49.54	43.85
Stage 2	70.42	44.26	36.07
Stage 3	70.42	44.26	21.97

Table 4.3: Chamfer distance (CD) on the validation set after pretraining for 240K iterations the first block (stage 1), second block (stage 2) and third block (stage 3). Lower is better.

	CD		
Vertices	156	628	**2466**
Stage 4	43.53	23.57	**15.62**

Table 4.4: CD on the test set after full training for 2.8M iterations. Lower is better.

Once all modules of the network are pretrained, our model is optimized for 2.4M iterations during more than 4 days. The final values of F-score and Chamfer distance on the test set are detailed in tables 4.5 and 4.4. Jointly optimizing all modules together also improves the performance of first and second blocks. We should always expect that to happen, since blocks that work with a higher number of vertices are going to pool more features from the image feature network and, therefore, optimize it in benefit of blocks that work with a lower number of vertices. At the end of third block we are obtaining a Chamfer distance of 15.62 (table 4.4), way lower than 22, and an F-score of 77.47% at threshold $\tau = 9$ cm. This means that if we consider two points to be equivalent if they are at most 9 cm away from each other, a little bit more than three quarters of our predicted mesh is equivalent to the ground truth mesh to predict, on average. If we take into account that ground truth point clouds usually contain the insides of cars (steering wheel, seats, etc)

we can conclude that the model is performing really well.

Threshold	$\tau = 5$ cm			$\tau = 9$ cm			$\tau = 18$ cm		
Vertices	156	628	**2466**	156	628	**2466**	156	628	**2466**
Stage 4	6.89%	23.28%	**45.56%**	28.86%	61.60%	**77.47%**	77.11%	91.56%	**95.02%**

Table 4.5: F-score (%) on the test set at different thresholds after full training for 2.8M iterations. Larger is better.

4.2.2 Qualitative results

In order to be able to assure that our model is showing true positive results, we always have to validate qualitatively that our results on the test set. The reason is that, as observed in Pixel2Mesh, most of the measures used to quantify the accuracy and performance of a 3D reconstruction model focus only on distance between point clouds and ignore other important properties like smoothness, uniformity, etc. Figure 4.2 gathers some examples of 3D models reconstructed by our model. We show the input image (cropped to make it easier to see), our predicted model and the ground truth model.

As it can be observed, the model successfully recovers most of the geometric details included in the original image, as well as "imagines" occluded parts of the car based on an internal classification between models that the network has seen. Hence, one might think that the network is first choosing the memorized model that most closely resembles the model shown in the image, and afterwards is adapting it to fulfill the geometric constraints imposed by the 2D projection of the mesh over the camera plane. We should remember, nevertheless, that models rendered in the test set have not been seen by the network during training or validation.

4.2.3 Failure cases

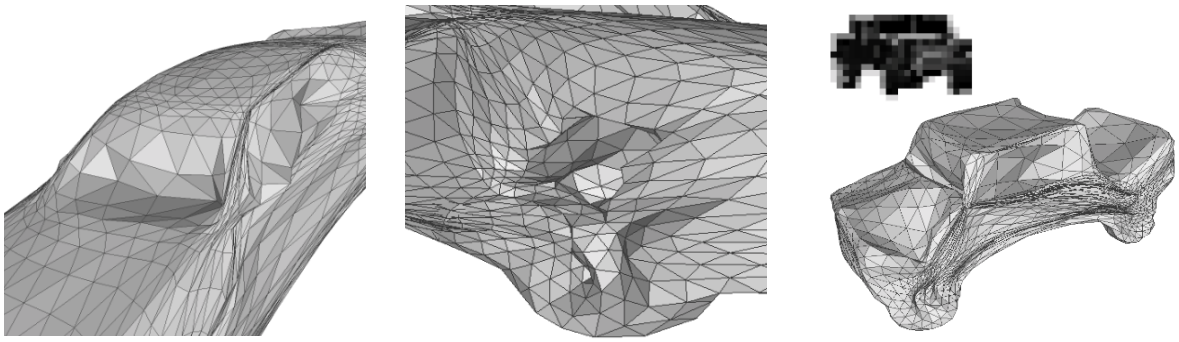
Even though our model is able to reconstruct very good 3D meshes for most of the models contained in the test set, it experiments some weaknesses when shaping certain kinds of surfaces, that are highly related to the kind of data we are working with. We should mention the following failure cases (see figure 4.1):

1. *Sunken windscreens* (figure 4.1a). Network optimization tends to sink car windscreens. The reason is that most of ShapeNet car models contain not only the exterior of the car, but also the inside. Therefore, one will find seats, the dashboard and the steering wheels inside, which possess high curvature and, thus, a large number of points are needed to describe it.

2. *Wrinkled wheels* (figure 4.1b). We can mainly attribute this problem to the large difference of concentration of points between wheels and the nose or the back of cars.

3. Failure to reconstruct *boxy cars* (figure 4.1c). This problem appears with cars that have a boxy shape. In this case, the network has learnt to concentrate all points of the nose and back of the car around the wheels, and found this is optimum. The reason this happens is because boxy cars consist of flat surfaces that are usually described by a few number of points. Therefore, most of the point are concentrated on any high curvature spot of the car, like the wheels.

All three problems share a common origin, which is a non uniform ground truth point cloud. In order to fix this, the easiest solution would be to resample ground truth points to be more uniformly distributed.



(a) Sunken windscreens (b) Wrinkled wheels (c) Boxy cars

Figure 4.1: Example of scenarios in which our model fails to reconstruct the right geometry: sunken windscreens, wrinkled wheels and boxy cars.

4.3 Applications

We already have discussed some applications of this model. We can summarize them as three main possible uses:

1. *3D mesh reconstruction.* Its main use could be reconstructing 3D meshes or point clouds of cars in an autonomous driving scenario, knowing 3D bounding boxes of cars. This model comfortably recovers 3D shapes of cars located very far from the camera (up to 35 meters away). As opposed to trying to classify among a list of different CAD models and adjust them to maximize the similarity between the original image and the projection of the chosen model (as in [27]), our model internally classifies and adjusts the shape to match the image feature perspective while doing it in a fully differentiable end-to-end fashion.

2. *Scene flow estimation* or *optical flow estimation.* After reconstructing an object as a solid 3D mesh, its movement can be described as that of a rigid object, that is, as a movement $T \in SE(3, \mathbb{R})$ (proper rotation plus a translation). Moreover, most of the time the rotation can be approximated by a rotation around an axis orthogonal

to the road plane, i.e., Y axis. This means that we can describe its movement with just 4 parameters (3 for translation vector and 1 for the rotation). This is way simpler than to attribute movement predictions to each of the pixels representing a certain object as if they did not belong to the same rigid object. Thus, after predicting a mesh for any pair of cars of consecutive points in time, one can easily infer the rigid transformation T relating them (simply comparing bounding boxes, or instead using Procrustes analysis [20]), and assign a scene flow

$$\text{scene flow of point } x = V(x) = Tx - x \tag{4.3}$$

to each point of the mesh x. For optical flow we would just project the scene flow over the camera plane as $w(x) = KV(x)$, where K is the camera matrix.

3. *3D mesh annotation.* Even though some KITTI [7] data has been annotated with the aid of CAD models, most real world driving datasets lack of "3D segmentation" or 3D meshes of vehicles present in the scene. Due to that most of models rely on synthetic datasets in order to be properly trained with a reasonable amount of data. However, using and adapting this tool to KITTI, for example, could easily boost the process of annotation of 3D meshes.

Figure 4.2: Qualitative results of our full trained model on the test set. We show the input cropped image (left), our prediction (middle) and ground truth (right).

Conclusion

With the present work, we intended to expose our developed approach for generating 3D shapes of cars from a single RGB image and the aid of a 3D bounding box. In it, we have explained how we have rendered a very large dataset with 300K samples for our task, and how we have adapted the end-to-end framework Pixel2Mesh [26] to incorporate 3D bounding boxes in the task of 3D shape reconstruction particularized for cars.

Due to the differences between the tasks addressed by both approaches, as well as the data used, we do not possess a fair comparison of quantitative results that we can use as a baseline. However, we show how our model obtains a presumably better performance in terms of quantitative measures like F-score or Chamfer distance, reaching a 77% of F-score with a threshold of 9 cm on our test set, which is comprised of car models that the network has not seen.

In addition, qualitative results on our test set demonstrate that our fully trained model successfully recovers most of the main geometric details of cars shown in images at very different levels of resolution, and fills the non-visible part of the car using its knowledge and experience from previous examples seen. We as well show how the network fails to predict the right 3D mesh reconstruction for certain spots of the car or certain kind of surfaces, due to the non-uniformity of the ground truth data used. Nevertheless, we also observe that these problems can be fixed by properly processing the ground truth point clouds of our dataset.

Even though the model successfully carries out the task its being designed for on the test set, both quantitatively and qualitatively, we list some further steps that we should follow to improve the performance of this model and transferability to real-world data:

1. First, in order to correct the failure cases in which the network fails to reconstruct certain surfaces properly, a resampling of all ground truth point clouds should be performed.

2. With the purpose o to assuring the right transferability of this model to real-world data (like KITTI [7]), more realistic shaders should be used to render vehicles in a very wide spectrum of lighting conditions. Otherwise, techniques of image-to-image translation can be applied to modify current rendered images lighting conditions.

3. In a typical autonomous driving scenario, we will usually have parts of the car occluded. Thus, our model should be able to deal with this occlusion and infer non-visible parts of the car. To do so, training data could be stochastically stamped with masks of real-world objects that arouse occlusion in the context of autonomous driving.

Finally, we must openly admit our satisfaction with the work developed and the pleasing results obtained. We also express our desire that improvements to this approach serve directly or indirectly as valuable tools for 3D reconstruction of vehicles and, ultimately, for autonomous driving development.

Bibliography

[1] Michael M. Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric Deep Learning: Going beyond Euclidean data. *IEEE Signal Processing Magazine*, 34:18–42, July 2017.

[2] CBInsights. Autonomy is driving a surge of auto tech investment. https://www.cbinsights.com/research/auto-tech-startup-investment-trends/.

[3] Angel X. Chang, Thomas A. Funkhouser, Leonidas J. Guibas, Pat Hanrahan, Qi-Xing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su, Jianxiong Xiao, Li Yi, and Fisher Yu. Shapenet: An information-rich 3d model repository. *CoRR*, abs/1512.03012, 2015.

[4] Christopher B. Choy, Danfei Xu, JunYoung Gwak, Kevin Chen, and Silvio Savarese. 3D-R2N2: A Unified Approach for Single and Multi-view 3D Object Reconstruction. *arXiv e-prints*, page arXiv:1604.00449, Apr 2016.

[5] Su H. Fan, H. and L.J. Guibas. A point set generation network for 3d object reconstruction from a single image. *CVPR (2017)*, 2017.

[6] Yuan Gao and Alan L. Yuille. Exploiting Symmetry and/or Manhattan Properties for 3D Object Structure Estimation from Single and Multiple Images. *arXiv e-prints*, page arXiv:1607.07129, July 2016.

[7] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[8] Ian J. Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Maxout Networks. *arXiv e-prints*, page arXiv:1302.4389, Feb 2013.

[9] Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, New York, NY, USA, 2 edition, 2003.

[10] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask R-CNN. *arXiv e-prints*, page arXiv:1703.06870, March 2017.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. *arXiv e-prints*, page arXiv:1512.03385, December 2015.

[12] Sergey Ioffe and Christian Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. *arXiv e-prints*, page arXiv:1502.03167, Feb 2015.

[13] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *arXiv e-prints*, page arXiv:1412.6980, Dec 2014.

[14] J. Krishna Murthy, G. V. Sai Krishna, Falak Chhaya, and K. Madhava Krishna. Reconstructing Vechicles from a Single Image: Shape Priors for Road Scene Understanding. *arXiv e-prints*, page arXiv:1609.09468, September 2016.

[15] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[16] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998.

[17] Moritz Menze and Andreas Geiger. Object scene flow for autonomous vehicles. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[18] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[19] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Neurocomputing: Foundations of research. chapter Learning Representations by Back-propagating Errors, pages 696–699. MIT Press, Cambridge, MA, USA, 1988.

[20] Peter Schönemann. A generalized solution of the orthogonal Procrustes problem. *International Conference on Computer Vision (ICCV)*, 1966.

[21] Thomas W. Sederberg and Scott R. Parry. Free-form deformation of solid geometric models. *SIGGRAPH Comput. Graph.*, 20(4):151–160, August 1986.

[22] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. *arXiv e-prints*, page arXiv:1409.1556, September 2014.

[23] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, January 2014.

[24] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going Deeper with Convolutions. *arXiv e-prints*, page arXiv:1409.4842, Sep 2014.

[25] Richard Szeliski. Computer vision algorithms and applications, 2011.

[26] Nanyang Wang, Yinda Zhang, Zhuwen Li, Yanwei Fu, Wei Liu, and Yu-Gang Jiang. Pixel2Mesh: Generating 3D Mesh Models from Single RGB Images. *ArXiv e-prints*, page arXiv:1804.01654, April 2018.

[27] Shunyu Yao, Tzu Ming Harry Hsu, Jun-Yan Zhu, Jiajun Wu, Antonio Torralba, William T. Freeman, and Joshua B. Tenenbaum. 3D-Aware Scene Manipulation via Inverse Graphics. *arXiv e-prints*, page arXiv:1808.09351, August 2018.

[28] Matthew D Zeiler and Rob Fergus. Visualizing and Understanding Convolutional Networks. *arXiv e-prints*, page arXiv:1311.2901, Nov 2013.